Xiwei Xu
Ingo Weber
Mark Staples

Architecture for Blockchain Applications

Springer

Architecture for Blockchain Applications

Xiwei Xu • Ingo Weber • Mark Staples

Architecture for Blockchain Applications

Springer

Xiwei Xu
Data61, CSIRO
Eveleigh, NSW
Australia

Ingo Weber
Data61, CSIRO
Eveleigh, NSW
Australia

Mark Staples
Data61, CSIRO
Eveleigh, NSW
Australia

ISBN 978-3-030-03034-6 ISBN 978-3-030-03035-3 (eBook)
https://doi.org/10.1007/978-3-030-03035-3

Library of Congress Control Number: 2018962552

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Foreword

This book provides an excellent overview of the engineering aspects of blockchains. You will learn what blockchains are, the current options for platforms, the application areas in which they may be used, and how do you, as a software engineer, design software to utilize blockchain technology. What I will do in this foreword is explore the disruptive nature of blockchains. Every popular article about blockchains mentions its disruptive nature. What does this mean?

Let me begin by discussing the general problem of technology transition. It is generally accepted in the technology transition community that it takes roughly 15 years from the inception of a technology to its broad adoption. Some technologies, notably the smart phone and the World Wide Web, have shortened that period and others have languished until the supporting infrastructure is ready for the technology. Containerization is a technology that existed for almost 30 years before Docker began and made it mainstream. Two important elements that help determine the time for a concept to become mainstream are the existence of educational materials such as this book and the publicity around the technology.

Although blockchains did not spring fully realized and built on prior work, the introduction of Bitcoin in 2009 can be considered the birth of blockchain. Since we are now a decade into its life, the 15-year estimate for widespread adoption seems to be proving out. As detailed in this book, preliminary applications are emerging beyond cryptocurrency. It seems reasonable that in another 5 years, blockchains will have entered the software engineering toolbox as a mature option with various offerings, and an engineer can apply the techniques discussed here to help design systems that utilize blockchains in some form.

Now let us turn to the disruptive nature of blockchains. Calling blockchains disruptive begs two questions: disruptive to whom and what does the disruption consist of?

Technologies that are disruptive to the consumer are actually quite rare. Smart phones and the World Wide Web are two of the most recent ones. These have changed the lives of almost everyone. The distinction between the disruption caused by the World Wide Web and the cloud is a useful example. One changed the lives of the consumers, and the other changed the lives of the producers.

The World Wide Web changed the manner in which people interact with each other, with businesses, and with government. The cloud, on the other hand, is an enabling technology that supports the expansion of the World Wide Web but is mostly invisible to the consumer. As a consumer, I do not care whether my news source is delivered from a cloud platform or a local platform. As a producer, however, I am concerned with production issues such as reliability, scalability, and cost. Consumers see an indirect impact for the cloud, but the disruption is primarily to the producers.

Blockchains in this dichotomy are disruptive to the producer, and their existence will be only indirect on the consumer. To use a supply chain example, as a supplier, I am happy to have a more efficient and reliable mechanism for me to get paid, but it is not disruptive to a great extent. One place where the use of blockchains disrupts the life of consumers is in areas where there is no functioning bank system. An example of this is the UN use of blockchains for refugees.[1]

In places with functioning government services, it is the producers who are potentially disrupted by blockchains. So let us dig more deeply into some of the use cases for blockchains and see where the disruption might occur.

- Supply chain. The current process for a supply chain that traverses international borders is that the producer and consumer agree on a price. The purchaser provides some proof of funds—for example, a letter of credit. The producer now produces the goods and ships the goods. They travel through several changes of responsibility and end up at the consumer. To see what is meant by changes of responsibility, consider a cotton grower in Australia who sells cotton to a consumer in Thailand. Cotton is priced in US dollars. The cotton grower loads the cotton on a train (one change in responsibility), the train goes to a port where it is loaded on a ship (another change in responsibility), the ship goes to a port in Thailand where it is loaded onto a truck (another change in responsibility), and the truck goes to the consumer where it is finally delivered. At this point, the cotton grower can cash in the letter of credit. Two points to mention. First, this process takes months, and the Australian dollar may have fluctuated relative to the US dollar, so the cotton grower is engaged in currency speculation. Secondly, each of the changes of responsibility is accompanied by entering information into at least one computer system, if not two. Any discrepancy between the relevant computer systems must be reconciled manually.

  How does this process change utilizing blockchains? It changes in two fashions. First there is only one source of truth—the blockchain. All of the participants have agreed to interact with the blockchain. Thus, there is no reconciliation necessary. Secondly, the consumer deposits the purchase money (in US dollars) into the blockchain, replacing the letter of credit. This allows the cotton grower to get incremental payments at each change of responsibility. This reduces the risk of currency fluctuation to the grower.

[1] https://www.technologyreview.com/s/610806/inside-the-jordan-refugee-camp-that-runs-on-blockchain/.

Now where is the disruption in this scenario? Because there is no need for reconciliation, the labour costs for reconciliation disappear. Secondly, letters of credit are difficult for banks to handle, as they involve much manual operation. Thus, the disruption consists of automating some processes that used to be manual. Easier, smoother, faster—yes. Disruptive? In the same sense that automation is disruptive to those displaced.

- Proof of identity. Currently, you acquire a proof of identity—e.g. driver's license or passport—by providing some other proof of identity, e.g. a birth certificate, to a trusted authority that then issues the proof of identity. You can carry this proof with you and produce it on demand.

  How will this change with blockchain? You prove your identity by providing some proof of identity—the UN in the example cited uses retinal scans—to a trusted authority that then enters your identity onto the blockchain. You can save this proof of identity on a smart card or retrieve it through some form of secure access. The individual or system that is interested in your identity will recover it from the blockchain, although the person or system checking your identity may be interested in some attribute of you that they can retrieve from the blockchain without the necessity of retrieving your identity.

  Where is the disruption here? Proofs of identity do not depend on physical papers such as passports, although they do depend on the ability to retrieve the proof of identity. You will not need multiple forms of identification; the blockchain will suffice, although this depends on all of the institutions that you interact with accepting the blockchain identification and being able only to retrieve information relevant to them. Disruptive? More than the supply chain in that there is a single source of identity and attributes for you. This will simplify life for you and, again, displace personnel involved in the production and verification of identification documents. It will also reduce the incidence of forgery of identification documents.
- As a final example, let us examine the problem of compliance. Financial institutions, e.g. banks, must comply with a variety of regulations. Each regulation is verified by individuals in some regulatory agency. Compliance is verified through an audit process. Individuals from the regulatory agency coordinate with individuals from the financial institution to perform an audit. This requires the individuals from the financial institution to collect the relevant information. The individuals from the regulatory agency will then examine the information provided to determine whether it conforms to the regulation. They will also do spot audits of the source of the information to determine that the provided information is, in fact, representative of the data that they are auditing. Errors and 'red flags' are identified by the regulators, and they work with representatives of the financial institution to resolve errors and determine processes to eliminate the 'red flags'.

  How will this change with blockchains? First, the auditors and the financial institution can both access the data from the blockchain with assurance that they are looking at the same data. The auditors will have extraction software that will produce the reports they need without them relying on the financial institution.

Since the auditors have real-time access to up-to-date information, they are able to produce the quarterly reports that they need to have. Since the blockchain provides a single source of truth for the information, automated systems can produce the reports either on the regulatory side or the financial institution side.

Where is the disruption here? Both regulators and financial institutions will find their work simplified. Since there is only a single source of truth, errors are reduced. Since the auditors will have real-time access to the blockchain, the delay in gathering the information is reduced. Again, the personnel involved in producing reports from the financial institution will be reduced, and the personnel involved in auditing will be reduced.

Examining other use cases enumerated in Chapter 1, my conclusion is that introduction of blockchain technology will speed up existing processes, reduce the labour involved in performing operations, and reduce errors. These advantages are substantial, but disruptive in the same way that smart phones or the World Wide Web have been? I do not think so.

Why then does every popular article about blockchain mention 'disruption'? First, there is the question of what could be achieved with existing technology. A blockchain is a distributed data base + encryption + immutability + stored procedures (in the form of 'smart contracts'). There is no inherent reason why existing distributed database systems cannot be extended to add cryptography and immutability as features. So, the disruption consists of replacing one technology with another. Better, faster, less error prone, but not necessarily life changing for any of the participants except for those displaced by the labour savings.

Secondly, let us go back to the compliance example. The World Economic Forum, when discussing the disruption caused by blockchain, has this to say about compliance[2]: 'Given no legal/regulatory precedent, establishing a shared arrangement between the regulator and [financial institutions] will be arduous'. They say something similar for all of the use cases they analyse.

In other words, to achieve the benefits touted from blockchain will require a degree of cooperation among institutions that has not yet been achieved and that could have been achieved with modifications of existing technology.

So why all the hype and discussion of disruption? Blockchains offer the opportunity to rethink financial systems and arrangements. This is a tough sell to top management who must sign off on the costs associated with redoing existing systems. A look back in history might be instructive in this regard.

Recall the year 2000 (Y2K) problem. The year 2000 was coming, and the problems it would cause were well known in advance. Suppose you are an IT manager in a bank in 1995. You know you have to fix the Y2K problem but you also know that your systems are dated and need to be updated. You have been telling management for years that the systems needed to be updated, but they have been unwilling to allocate the money. Now you have a forcing function. You need to

[2]World Economic Forum, The Future of Financial Infrastructure, p. 99.

update your systems by the year 2000 or your bank will not be able to function. Management caves.

Now fast forward 20 years. The last time you updated your systems was 1999. They are again dated and badly in need of being updated. Your management again is reluctant to spend the money necessary. Now comes blockchain. Your argument is that all of the organizations with which you are interacting will be moving to blockchains and your bank needs to do this in order to function in the global market. More than this, there will be cost savings from the automation of tasks that are currently manual. Your argument is not as compelling as the year 2K argument, but it still carries weight with your management partially because of the publicity surrounding blockchains. Management is in the process of providing the financing for updating and rethinking your systems.

As a software engineer, you are on the receiving end of questions and instructions to convert your systems to use blockchains. Hence, you should read this book since it will provide answers to the questions and instructions for the conversion. You should look for opportunities both to deepen your understanding about blockchains and to apply them in situations where they provide the correct set of functionalities that you need for your particular problem.

Pittsburgh, PA, USA Len Bass
August 2018

How to Read This Book

When we first learned about Ethereum and its smart contracts, we were thrilled about the world of possibilities enabled by blockchain technology. As researchers, we have worked in this area from around mid-2015, which also marked the genesis of the public Ethereum blockchain. In our project work with startups, corporates, and government agencies, and in many interactions with the community, we found that the knowledge, tools, and methodologies for tapping into that potential were lacking. Therefore, we started investigating the issues in our core research: what do architects and developers need to build applications on blockchain? The result is this book, based on a stream of our earlier research publications, tools, and projects.

This book is primarily written for developers, software architects, and CIOs (Chief Information Officers), as well as students and researchers in these areas. The book captures the architectural view on software systems that use blockchain. It provides guidance on assessing the suitability of blockchain, on designing blockchain applications, and on assessing different architecture designs and trade-offs. The book is also a reference for blockchain design patterns and design analysis and refers to practical examples of blockchain-based applications.

This book is not a step-by-step tutorial on coding for blockchain, although the case study chapters contain code samples where these provide added insights. Instead, we focus on the bigger picture, the concepts, and technical considerations in the design of blockchain-based applications. We also limit the use of mathematical formulas except where they are critical, for cost estimation.

Readers who are familiar with particular platforms can easily skip that background, and also the initial example use cases which are more for illustration. Because we have drawn on our previous publications, there are a number of experiments that are included in the book. These experiments are similar to practical benchmarking and design studies that might be conducted by system architects, but in case the exact results matter less to you, you can jump over those sections easily. At the end of each chapter, we include a section called Further Reading, where references to additional material and the relevant literature can be found.

The book is structured into four parts, starting with the background. The introduction gives an overview of the issues discussed throughout the book, and

motivates the use of blockchain. It also contains a number of textual definitions of the most important terms, and a non-technical explanation of blockchain, in case you have to explain it to your parents, partner, or kids.

Chapter 2 gives background on existing blockchain platforms. We start, of course, with Bitcoin, and describe Ethereum. For enterprise use, several blockchain platforms have emerged, and we describe Hyperledger Fabric as an example of that class. The chapter closes with an overview of other platforms.

There are many varieties of blockchain platforms and possible configurations. In order to help the architect navigate this space, we describe the main dimensions and their implications for non-functional properties in Chapter 3.

In Chapter 4, we provide four use case examples of blockchain-based applications to convey a concrete understanding of how blockchain can be used to solve real-world issues. The domains of these use cases are supply chains, government registries, international money transfers, and electricity provision.

In Part II, we focus on the functional part of software architecture. We start with the main roles blockchain can play in an architecture in Chapter 5. Blockchain can be used as a data store, a computational element, a communication mechanism, and to manage assets and exert control. We also discuss considerations for integrating blockchain into a bigger system design.

Chapter 6 describes the design process, starting with the question of suitability: when should you use blockchain and when should you not? Once you settled on using blockchain, we then discuss how to make important decisions, such as what functionality to provide *on-chain* and what *off-chain*.

Making good use of blockchain in systems often requires solutions that are non-obvious, especially when starting out in this area. Chapter 7 provides a catalogue of 15 design patterns with in-depth descriptions, which have proven valuable in practice.

Due to specific properties of blockchain technology, model-driven engineering (MDE) is particularly amenable for blockchain-based applications. Chapter 8 describes two MDE methods, one for business processes and one for registries of assets.

Part III covers the non-functional aspects of blockchain applications, which are often cross-cutting concerns. Cost and cost estimation are discussed in Chapter 9. Similarly, Chapter 10 discusses performance, with a focus on latency. In both cases, estimates allow understanding the implications of a particular choice of platform, parameters, or blockchain configuration.

Dependability and security are discussed in Chapter 11. These two topics are related to six properties: confidentiality, integrity, safety, maintainability, availability, and reliability. We include some insights from observing the Ethereum and Bitcoin blockchains in this chapter, which architects and developers should consider when designing and building applications.

In Part IV, three use cases give practical insights. AgriDigital describe their experiences from three supply chain pilots in Chapter 12, with a focus on reducing the counterparty risk in supply chains. SecureVote developed a blockchain-based voting solution, which they describe in Chapter 13. This system runs in production

on the public Ethereum blockchain. originChain's use case in Chapter 14 is also on supply chain, but specifically targets provenance tracking in international trade.

Finally, in the Epilogue we reflect on the contents of the book and its major points. There we also speculate on the role blockchain and its applications can play in the future.

Acknowledgements

Writing a book is a long journey, which requires the support of many. We want to thank the contributors to individual chapters, Cesare Pautasso, Qinghua Lu, Alex Ponomarev, An Binh Tran, Paul Rimba, Rajitha Yasaweerasinghelage, Sin Kuang Lo, Ralph Holz, and Vincent Gramoli, as well as Bridie Ohlsson, Katherine Davison, and Emma Weston from AgriDigital, and Max Kaye and Nathan Spataro from SecureVote. We further thank members of the blockchain community in Sydney and generally in Australia, and numerous academics globally, for the many discussions that helped us shape and sharpen our thinking on this complex topic.

In this book, we draw on a number of our previous publications. Thanks go to our co-authors (in alphabetical order): Alex Ponomarev, An Binh Tran, Bin Liu, Cesare Pautasso, Guido Governatori, Jan Bosch, Jan Mendling, Len Bass, Liming Zhu, Paul Rimba, Qinghua Lu, Régis Riveret, Rajitha Yasaweerasinghelage, Ralph Holz, Shiping Chen, Sin Kuang Lo, Vincent Gramoli, Xiao Liang Yu, and Yin Kia Chiam. We also want to thank the anonymous reviewers of our papers and Tim Wellhausen, the shepherd for the design patterns paper, for their helpful comments.

We are also grateful to the management of Data61, CSIRO, for their support and encouragement. Deep thanks go to our families for their endless patience in this long effort—which was long enough for two out of three authors to have a baby each. Finally, we want to thank Len Bass for writing an insightful foreword.

Legal Disclaimer for Code Samples

This book contains a number of code samples, in the following referred to as 'SOFTWARE'.

The SOFTWARE is provided 'as is', without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose, and noninfringement. In no event shall the authors, contributors, or copyright holders be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the SOFTWARE or the use or other dealings in the SOFTWARE.

Contents

Part I
Blockchain in Software Architecture

Chapter 1
Introduction

1.1 What Is Blockchain and Why Should I Care?

Blockchains are an emerging digital technology that combine cryptography, data management, networking, and incentive mechanisms to support the checking, execution, and recording of transactions between parties. A blockchain ledger is a list ('chain') of groups ('blocks') of transactions. Parties proposing a transaction may add it to a pool of transactions intended to be recorded on the ledger. Processing nodes within the blockchain system take some of those transactions, check their integrity, and record them in new blocks on the ledger. The contents of the blockchain ledger are replicated across many geographically-distributed processing nodes. These processing nodes jointly operate the blockchain system, without the central control of any single trusted third-party. Nonetheless, the blockchain system ensures that all nodes eventually achieve consensus about the integrity and shared contents of the blockchain ledger.

Transactions between parties such as payments, escrow, notarization, voting, registration, and process coordination are key in the operations of government and industry. Traditionally, these transactions are supported by trusted third-parties such as government agencies, banks, legal firms, accounting firms, and service providers in specific industries. Blockchains provide a different way to support these transactions. Instead of trusting third-parties, we would trust the collective jointly operating the blockchain and the correctness of their shared technology platform.

Blockchain technology was originally used for the Bitcoin digital currency, but blockchains are now being implemented in many other platforms and used for many other purposes. Just like a traditional database, a blockchain can in principle be used to represent transactions or information in any kind of application domain. But blockchains are different from traditional databases in important ways. These differences impact the design of systems that use blockchain.

X. Xu et al., *Architecture for Blockchain Applications*,
https://doi.org/10.1007/978-3-030-03035-3_1

The successful operation of a blockchain system relies on several key elements, including:

- Appropriate integrity criteria to be checked for each transaction and block
- The correctness of the system's software and technical protocols
- Strong cryptographic mechanisms to identify parties and check their authority to add new transactions
- A suite of incentive mechanisms to motivate processing nodes to participate in the community and to behave honestly, in their interests

For the software architect and engineer, blockchains are exciting because they can be used as a new foundation for re-imagining systems. They form a neutral infrastructure for processing transactions and executing programs. That is of potential interest for innovation at all touch-points between organizations or individuals. As such, *blockchain applications have the potential to disrupt the fabric of society, industry, and government*. Blockchains can also be used as a technology platform to handle some of the hard issues of data replication and system state synchronization with high integrity.

A Non-technical Explanation of Blockchain by Analogy

Imagine that a group of people, say the population of your community, want to introduce a special community currency. Let's call this currency the Community Dollar C$, which will be a noncash virtual currency. Initially everyone gets C$ 100, and everyone starts a physical ledger book where they note these holdings. The goal is to keep track of C$ ownership in all these ledger books, by ensuring the ledger books of everyone contain the same information.

Say, person *A* wants to pay C$ 50 to person *B*. Therefore, *A* asks everyone in the community to add that transaction to their ledger. Everyone checks if A has the money and signed the transfer order. If so, the transaction is added to the ledger. This results in an updated state where *A* has C$ 50 and *B* owns C$ 150.

Now the Community Dollar is starting to become popular, and many people use it. We start by grouping transactions onto paper pages, and rather than agreeing on each transaction individually, the whole community needs to find agreement which page to include. (Pages correspond to blocks in the blockchain.) Everyone still checks every transaction. To ensure that no one claims a transaction did not happen, we introduce cryptographic hashes that make sure no one can go back on the agreed set of ledger pages and the transactions on them. Assume we also have an incentive mechanism that encourages community members to stay honest and process transactions. That is needed to make such a decentralized system work. All of this can then happen without a trusted third-party, purely operated by members of the community.

1.1.1 Defining Blockchain

Before delving further into the details of the technology, we first define the main concepts. Blockchains maintain a ledger and implement a specific kind of distributed ledger technology.

Definition 1 (Distributed Ledger) A *distributed ledger* is an append-only store of transactions which is distributed across many machines.

Being 'append-only' is important: new transactions can be added, but old transactions cannot be deleted or modified. A new transaction might reverse a previous transaction, but both of them remain part of the ledger to allow auditability and ensure long-lasting integrity. We define the *concept* of a blockchain as follows.

Definition 2 (Blockchain) A *blockchain* is a distributed ledger that is structured into a linked list of *blocks*. Each block contains an ordered set of transactions. Typical solutions use cryptographic hashes to secure the link from a block to its predecessor.

A graphical representation of this concept is shown in Fig. 1.1. Cryptographic hashes ensure that a previous block cannot be changed. If the previous block was changed, its new hash would not match the originally recorded hash, so the link between the two blocks would break. We explain this mechanism in more detail in the next chapter, where we discuss specific blockchain platforms.

Some ingredients are necessary for the blockchain concept to work in practice as a system.

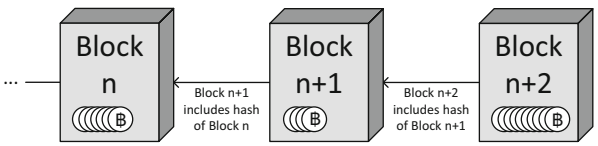


Fig. 1.1 Blockchain data structure

Definition 3 (Blockchain System) A *blockchain system* consists of:

(i) a *blockchain network* of machines, also called *nodes*;
(ii) a blockchain data structure, for the ledger that is replicated across the blockchain network. Nodes that hold a full replica of this ledger are referred to as *full nodes*;
(iii) a network *protocol* that defines rights, responsibilities, and means of communication, verification, validation, and consensus across the nodes in the network. This includes ensuring authorization and authentication of new transactions, mechanisms for appending new blocks, incentive mechanisms (if needed), and similar aspects.

For the verification of transactions, consider the example of Alice spending 2 Bitcoin (BTC), by transferring them to Bob. The system needs to ensure that the party initiating the transaction has Alice's authority and that Alice has the 2 Bitcoin available.

The above definition is still relatively broad and can capture blockchains of various sizes, degrees of openness, for various purposes, etc. The most well-known blockchains are Bitcoin and Ethereum, which are *public blockchains*.

Definition 4 (Public Blockchain) A public blockchain is a blockchain system that has the following characteristics:

(i) it has an *open network* where nodes can join and leave as they please without requiring permission from anyone;
(ii) all full nodes in the network can *verify* each new piece of data added to the data structure, including blocks, transactions, and effects of transactions; and
(iii) its protocol includes an *incentive mechanism* that aims to ensure the correct operation of the blockchain system including that valid transactions are processed and included in the ledger and that invalid transactions are rejected.

Public blockchains are often open leaderless peer-to-peer systems that manage the ownership of assets of value. Examples of such assets on Bitcoin and Ethereum blockchains are Bitcoin (BTC) and Ether (ETH) cryptocurrencies and digital tokens. In a public blockchain, there is not a high degree of trust in information from other nodes. Therefore, all full nodes verify everything, to reduce the risk of integrity violations jeopardizing the value of their own work. While this leads to redundant computation across the network, it is a direct consequence of the community of nodes collectively safeguarding the integrity of the blockchain.

In other settings, for example, within a large enterprise or in a consortium of companies, all blockchain nodes might be known and governed by other organizational or contractual mechanisms. These applications can be served by adopting a more relaxed trust assumption.

Finally, we define the term blockchain platform, which refers to the software used to run a blockchain.

Definition 5 (Blockchain Platform) A *blockchain platform* is the technology needed to operate a blockchain. This comprises the blockchain client software for processing nodes, the local data store for nodes, and any alternative clients to access the blockchain network.

Note that any blockchain platform must have client software with which processing nodes can operate the network, including for transaction propagation and block creation. Light clients may additionally exist, e.g. to enable mobile devices to read and write transactions to the network; these typically do not hold a full copy of the blockchain data structure. Alternative clients, both for processing and light nodes, may exist, particularly if the protocol is specified well.

1.1.2 Smart Contracts and Decentralized Applications

The transactions stored on a blockchain can be more than simple records of the exchange of assets—emerging blockchain systems also allow computer programs to be stored and to execute as part of transactions on the ledger. These are often called 'smart contracts', although the programs are typically not very smart and are often not related to legal contracts.

Definition 6 (Smart Contract) *Smart contracts* are programs deployed as data in the blockchain ledger and executed in transactions on the blockchain. Smart contracts can hold and transfer digital assets managed by the blockchain and can invoke other smart contracts stored on the blockchain. Smart contract code is deterministic and immutable once deployed.

The Bitcoin blockchain allows only very simple forms of smart contracts, but other blockchains such as Ethereum allow computer programs to be written in a 'Turing complete' language, that is, in principle, as expressive as every other general purpose programming language. As a result, blockchains can be more than a simple distributed database—they can be *general computational platforms*—albeit currently with severe practical limitations on computational complexity. This

capability significantly expands the power of blockchain systems and increases their range of use and potential for innovation.

Smart contracts can be used to administer the ownership of assets represented by the blockchain cryptocurrency or by digital token implementations using a smart contract—more on that below. Although smart contracts are not always used for legal contracts, they can sometimes be used to automate or monitor the execution of parts of legal contracts. Smart contracts can also implement games, bets, or lotteries. They can also define a protocol of interaction between different parties, like in a collaborative business process across companies, and can support many more use cases. Throughout the book, you will find many applications of blockchain that are only possible due to the smart contract capability.

Applications can be designed to provide their main functionality through smart contracts. Such applications are called *decentralized applications* or *dapps*, and we will discuss them in more detail in Section 2.2.5. *Tokenvote*, the system described in Chapter 13, is an example of a dapp. In this book, we generally talk about *blockchain-based applications*, i.e. applications that make significant use of blockchain. This includes dapps but is not limited to them—significant portions of such applications can be based on traditional systems.

1.1.3 Cryptocurrencies and Tokens

Cryptocurrencies are the base currencies of blockchains. *Ether* is the currency of the public Ethereum blockchain, and Bitcoin is the currency of the public Bitcoin blockchain (thereby highlighting a source of confusion due to overloaded terminology). The respective blockchain keeps track of the ownership of portions of that currency. Say, Alice owns 2 Ether and announces a transaction to transfer 1 Ether to Bob, offering a fee of 0.01 Ether. Once the transaction is included in a block mined by Charly, Alice has 0.99 Ether, Bob has 1, and Charly received the fee of 0.01 Ether. The sum of the money is not changed by these transactions, but the ownership of portions of it is.

Fees for transaction inclusion are paid in the base currency of a blockchain, although the client can choose to offer a fee of 0 (typically reducing the speed and/or likelihood of inclusion). Fees often relate to the *size* of a transaction, not its *value*: more data (including larger smart contracts to be deployed) incur higher fees. Similarly, more complex computations as a result of smart contract invocations incur higher fees. Transfers of 0.01 Ether incur the same fees as transfers of 100 Ether.

Digital *tokens* can be created and exchanged on blockchains. Usually tokens are created using smart contracts. Similar to a cryptocurrency, each token is controlled by an actor on the blockchain. Tokens might represent shares in a company, the right to benefit from future earnings, or perhaps virtual gold in an online game. The use of tokens has become widespread, and tokens can be seen as the first ‘killer app’ of using blockchain for things other than cryptocurrency.

1.2 Blockchain-Based Applications

Bitcoin has been operational since 2009, and its digital currency had a peak market capitalization of about US$335B in December 2017. The next-largest blockchain, Ethereum, had a market capitalization of US$138B in the same time frame, and there are many other small public blockchains with their own digital currencies. Private blockchains are increasingly deployed inside large enterprises and across industry consortia. The wide array of interest in blockchain technology is underlined by the fast evolution of its ecosystem, including easier deployment through Blockchain-as-a-Service, e.g. from Microsoft Azure[1] and IBM.[2]

Many banks are involved in trials of blockchain technology, including through the R3[3] or Ripple[4] organizations, which are applying blockchain to trade finance and cross-border payments. Financial transactions are the first, but not the only use case being investigated for blockchain technology. A blockchain implements a distributed ledger, which can in general verify and store any kind of transactions. Globally, many financial services companies, enterprises, startups, and governments are exploring its applications in areas as diverse as supply chain, electronic health records, voting, energy supply, ownership management, and protecting critical civil infrastructure. New businesses and business models are expected to arise, but as yet there are not a lot of examples of significant use in production of blockchain systems within industry or government.

Blockchains, particularly public blockchains, offer opportunities for disruptive innovation when implementing decentralized applications. Blockchains provide a new basis for trust in relationships in society, which can allow existing trusted third-party organizations to be disintermediated. In economies where trusted third-parties are not always trustworthy, a significant benefit of blockchain systems may be in the support they can provide for immutability (not changing prior records on the ledger) and non-repudiation (not being able to disown prior actions on the ledger). In developed societies, trusted third-party organizations are usually trustworthy, so the benefits of using blockchain technologies would instead likely arise from enabling faster business model innovation, reducing the cost of establishing business relationships and mitigating risks, and perhaps by reducing the cost or risk of transactions.

For applications of blockchain, there are two *categorically different types*: (1) does the blockchain hold the default source of truth, or (2) does it hold a (possibly incorrect) view of reality? Cryptocurrency is a case of the former: if Bob's account on the blockchain holds 1 Ether, he can control that. By default he is the owner—although a court might determine that he did not fulfil his part of an agreement

[1] https://azure.microsoft.com/en-us/solutions/blockchain/.

[2] http://www.ibm.com/blockchain/.

[3] http://www.r3.com/.

[4] http://www.ripple.com.

and has to pay the 1 Ether back to Alice. In the traditional world, there are some examples of things whose existence and ownership rely on database entries, such as land ownership rights, companies, and patents. These could be ported to a blockchain application of the first type. In contrast, a blockchain record of a physical asset and its state (location, quality, temperature, etc.) is an example of the second category. The view of the asset could be outdated, incorrectly measured, or wrong in some other way. As such, blockchain applications of the first type tend to be more straightforward in their implementation, although they require higher buy-in from the adopters due to their higher degree of reliance on a relatively new piece of technology.

We preview some application areas below and describe some particular use cases in Chapter 4. Three case study chapters in Part IV give detailed accounts from the industry.

1.2.1 Enterprise and Industry

Blockchains were first used for cryptocurrency but are now being used for many other purposes. The full potential of blockchain technology is likely to be realized outside financial services and government. Blockchains are a foundational horizontal platform technology that could be used in any industrial sector including agriculture, utilities, mining, manufacturing, retail, transport, tourism, education, media, healthcare, and the sharing/P2P economy. Generic applications in these sectors include:

Supply Chain When tracking physical assets through changes in ownership and handling, key events and agreements can be recorded and communicated through data stored on a blockchain. This results in provenance information for goods and can provide improved logistics visibility and supply chain quality. Key events within the supply chain could also be linked to automatic payments with the use of smart contracts. Supply chain cases are also captured in the use case chapters on AgriDigital (Chapter 12) and originChain (Chapter 14) as well as Section 4.1.

Internet of Things (IoT) Storage, Compute, and Management Devices connected to the Internet can use the blockchain as a persistent and highly available storage solution. They can also use smart contracts to provide a global distributed-computing capability and can rely on the blockchain as a secure channel for receiving information about software and configuration updates and dynamically-delegated access control. This can include physical access control, for locking and unlocking devices.

Metered Access to Resources and Services Monitoring and payment for usage of utilities or services can be provided by IoT devices and associated smart contracts. An electricity use case is described in Section 4.4.

Digital Rights and IP Management A blockchain can provide a trusted registry of media assets or other intellectual property and can provide the ability to manage, delegate, or transfer access and rights information for those assets. Note that media are not necessarily stored on the blockchain itself. Instead, cryptographic hashes, metadata, and other identifiers stored on the blockchain might be integrated with bulk off-chain storage and communication technologies.

Data Management A blockchain can create a metadata layer for decentralized data sharing and analytics. Although large datasets themselves are unlikely to be stored on it, a blockchain can help to discover and integrate those datasets and data analytics services. Access control mechanisms implemented on a blockchain may allow public data sources to be integrated more easily with private datasets and analysis services. See also Section 4.2 for a use case on open data.

Attestation and Proof of Existence A blockchain can be used to record evidence of the existence of data or documents, by creating a timestamped record of a cryptographic hash of the contents of those documents. This can be combined with records of the attestation or witnessing of corresponding physical documents by trusted third-parties. However, it can be significantly harder to demonstrate the uniqueness or non-existence of such document records, unless there is a widely accepted strict normal form for their contents.

Interdivisional Accounting Multinational companies or large enterprises with separate divisional business units often have jurisdictional or governance needs to control their own internal accounting but also share accounting information with other divisions. A straightforward application of blockchain technologies on a shared private network can create a shared distributed ledger of interdivisional accounts at the interfaces between divisions.

Corporate Affairs (Board and Shareholder Voting and Registrations) The voting authorities of board members or shareholders in companies can be recorded and proxied on a blockchain. Smart contracts on blockchains can use that record to adjudicate votes conducted on the blockchain for specific motions. As blockchain transactions are not necessarily hidden, cryptographic mechanisms may be required to prevent potentially undesirable strategic voting behaviours. The company SecureVote describes their approach and architecture in Chapter 13.

1.2.2 Financial Services

Financial services applications using blockchain technology may include:

Digital Currency New forms of money can be implemented on blockchains, but these can also serve as a foundation for incentive models that support integrity for many blockchain systems. Blockchains allow digital currency to be transferred between parties, often without those transfers being processed or recorded by banks

or payment services. With smart contracts, blockchains may be able to support 'programmable money', where automatically enforced policies are attached to specific parcels of currency.

(International) Payments Can be facilitated by blockchain, often via digital currency with local exchanges between the digital currency and fiat currencies. Public blockchain cryptocurrency payments are usually *pseudonymous*. For example, on Bitcoin, transacting agents (which are not necessarily persons) are only identified with a cryptographic key. Therefore international exchange of the Bitcoin digital currency can be performed without establishing real-world identity, and we may not know which actual person is behind which account. Nonetheless, international payments usually have regulatory requirements to establish the identity of participants, as part of Anti-Money Laundering (AML) and Counter-Terrorism Financing (CTF) regulations, and AML/CTF requirements are not obviated by the use of a blockchain. Still, transacting parties can choose to establish their real-world identities to each other and to local exchanges, and this is typically how regulation of blockchain-based international payments is enforced. This topic is also covered in Section 4.3.

Reconciliation for Correspondent Banking Reciprocal nostro/vostro accounts can be replaced by a single shared ledger. Rather than conducting laborious end-of-day reconciliation as a batch task, the two banks can create a single shared view of truth between their accounts, maintained in real time. To limit the distribution of this commercially sensitive information, usually the shared ledger would be restricted to just the two correspondent banks concerned.

Securities Settlement The joint exchange of payment and security holdings is enacted as a single transaction on a blockchain. The exchanged assets are typically represented by tokens implemented on the blockchain, either using smart contracts or other asset representation capabilities provided by the underlying blockchain platform. Payments too are sometimes made using such tokens standing for conventional fiat currency or can sometimes be made using the native cryptocurrency on the blockchain.

Markets Smart contracts on blockchains can provide a platform for making and accepting offers to trade assets or services. The blockchain will record the status of these trade offers. Individual smart contracts could themselves carry the digital currency required to be paid on fulfilment of these offers. This functions as a kind of escrow, without the need for a trusted third-party organization. However, today's blockchain systems are not suitable for high-frequency (low latency) market trading. Also, for public blockchains, pending transactions are visible across the network which can allow a kind of 'front-running', where participants (here, usually the nodes operating the blockchain) might unfairly take advantage of information in these as-yet unexecuted instructions.

Trade Finance The blockchain can be used to evidence trade-related documents in order to reduce lending risk and improve access to finance for industry. Smart

contracts could control inter-organizational process execution (see also Chapter 8) and transparently automate delayed or instalment payments. This can improve assurance about previous trading history and about current commitments by counterparties, which can reduce risk to trade finance providers, thus allowing more widespread and economical trade finance offerings into the market. AgriDigital's case study chapter, Chapter 12, discusses these issues.

1.2.3 Government Services

Blockchains could target improved government service delivery, and private blockchains could be used to facilitate information sharing and process coordination across agencies within government. Application areas being explored in governments globally include:

Registries and Identity Including the identities and attributes of persons, companies, or devices, licensing, qualifications, and certifications. Storing registry entries or cryptographic certification of registry entries on a blockchain can facilitate access to and validation against the register. Blockchains could be used to share authenticated identifiers for individuals and companies, and these identifiers could in turn also enable many other blockchain applications. Blockchains can support federated management of multiple related registries, by allowing different agencies to retain authoritative control over the contents of their registers, but still provide a shared view of truth about how their registers are interrelated—see also Chapter 8. The contents of some government registers are public, but in general there are often complex considerations about privacy and confidentiality.

Grants and Social Security Smart contracts could automate the process coordination to apply for, decide on, and distribute payments for grants and social security. With a sufficiently sophisticated payment environment, a smart contract could automatically limit payments to approved suppliers or categories of expenses. One early use of blockchain in this way was to account for allowances and payments by refugees in a UN refugee camp. Other experiments have been carried out in the context of disability support grants.

Quota Management Government-granted quotas, allocations, and rights to physical resources could be awarded and tracked through tokens established on a blockchain. Examples include water access licences providing rights to take a certain volume of water from specific sources during specific time frames or CO_2 emission credits. Where policy allows, blockchain could support an independent secondary market for these rights. The blockchain creates an ongoing immutable audit log of these rights and their use.

Taxation Possible applications range from automated collection of tax using smart contracts to improved compliance by authoritative publication of taxation regulation and calculation tools as smart contracts on blockchain.

1.3 Blockchain Functionality

Software architects need to understand functional and non-functional characteristics of blockchains. In this section, we sketch the functionality of blockchain as a data store and as a computational infrastructure. Figure 1.2 gives an overview of the functionality a blockchain can offer. Blockchains are complex, network-based software components, which can provide data storage, computation services, and communication services. Blockchain features can include cryptographically secure payment, mining, transaction validation, incentive mechanisms, and permission management. What is called an *oracle* supplies information about the external world to the blockchain, usually by adding that information to the blockchain as data in a transaction. Below we expand on the two major functional capabilities of blockchain, for data storage and for computation.

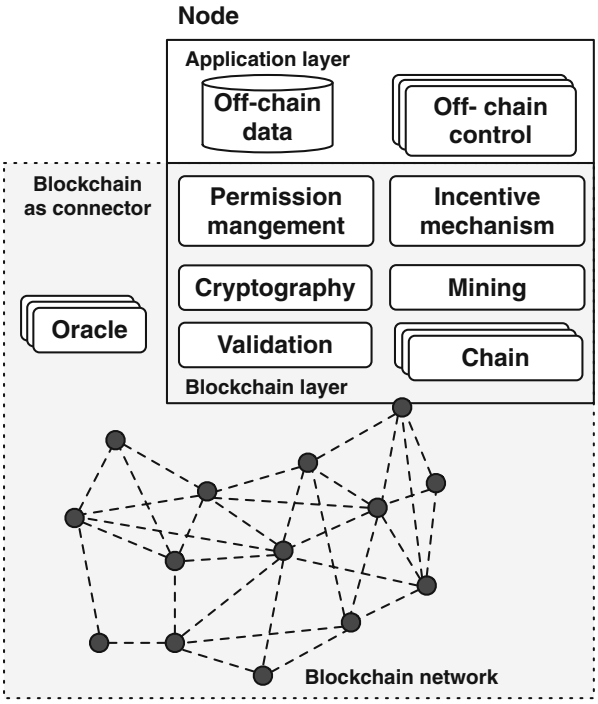


Fig. 1.2 Overview of the functionality that blockchain can offer as an architecture element. © 2016 IEEE. Reprinted, with permission, from Xu et al. (2016)

1.3.1 Blockchain as Data Storage

As a data structure, a blockchain is an ordered list of blocks, where each block contains a small (possibly empty) list of transactions. Each block in a blockchain is 'chained' back to the previous block, by containing a hash of the representation of the previous block. Thus historical transactions in the blockchain may not be deleted or altered without invalidating the chain of hashes. Combined with computational constraints and incentive schemes on the creation of blocks, this can in practice prevent tampering and revision of information stored in the blockchain. As a data storage facility, information in a blockchain is recorded within the transactions and within blocks. Important categories of information are transactions about cryptocurrency and transactions involving tokens for other kinds of assets.

Transactions

Transactions update the state recorded on a blockchain. For cryptocurrency transactions, the state information is about the transfer of holdings of cryptocurrency between accounts (addresses). Sometimes additional data can be recorded with a transaction which might have meaning for participants or systems outside of the blockchain. On blockchains such as Ethereum, transactions can record code, variables, and the results of function calls. Public key cryptography and digital signatures are normally used to identify accounts and to ensure integrity and authorization of transactions initiated on a blockchain.

A simplified life cycle of transactions is shown in Fig. 1.3. Once created, the transaction is signed with the signature of the transaction's initiator, which provides the authorization to spend the money, create a contract, or pass the data parameters associated with the transactions. A signed transaction should contain all the information needed to be executed.

A proposed transaction is sent to a node connected to the blockchain network, which checks the validity of the transaction. Invalid transactions are discarded. Valid transactions that are previously unknown to the node are propagated to other connected nodes. These will in turn further validate the transactions and send them to their peers, until the transactions reach every node in the network.

In a global network, this flooding approach means that a valid transaction will usually reach the whole network within a few seconds. To ensure that the transaction propagates, senders do not need to trust any individual node they send the transaction to, as long as they send it to enough other nodes. Recipients do not need to trust senders, because all transactions are signed and can be independently validated by any node.

When a transaction reaches a 'mining' node, it is verified and may be included in a block. *Mining* is the process of appending new blocks to the blockchain data structure. A blockchain network relies on *miners* to aggregate valid transactions into blocks and append them to the blockchain. New blocks are broadcast across the whole network, so that each full node holds a replica of the whole ledger.

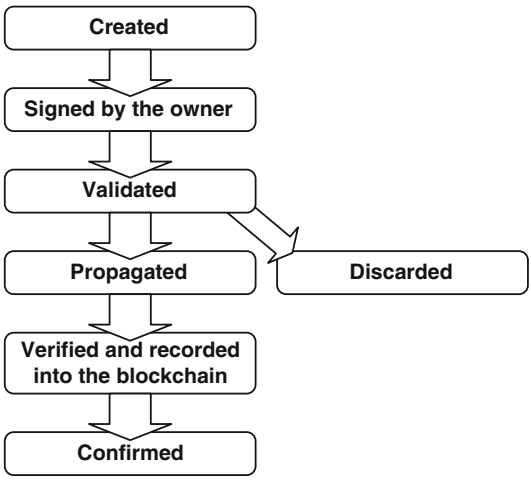


Fig. 1.3 Simplified transaction life cycle

The network aims to reach a consensus about the latest block to be included into the blockchain. There are different consensus mechanisms, e.g. 'proof of work' or 'proof of stake', which we will describe in more detail later.

However, there is no certainty about whether a particular transaction will eventually be committed or whether it will be outdated. An outdated transaction will be considered an invalid transaction forever, e.g. due to an alternative transaction getting committed. Also, it is often impossible to know whether a transaction that is invalid in some state of the system could ever become valid in a later state. For some consensus mechanisms, the inclusion of the transaction in a newly mined block on some branch of the chain is not sufficient to guarantee that a transaction is permanently added to the blockchain: if the blockchain forks (i.e. conflicting versions of new blocks are proposed simultaneously), then the block comprising the transaction may simply be discarded, in which case it could be re-included later. Each of these inclusions takes time as they require the system to solve computationally hard cryptographic puzzles. If during that time a conflicting transaction Tx' is included first, then the original transaction Tx may simply become invalid, e.g. until a possible third transaction Tx'' compensates for the effect of transaction Tx'.

Depending on the consensus mechanism and the required guarantees, different blockchain platforms and applications can have different notions of when a transaction is *committed* or *confirmed* and thus be immutable. For example, in Bitcoin users often wait five subsequent blocks to be appended to the block containing a transaction before viewing the transaction as committed. However, this is a probabilistic commitment, and so the number of blocks one should wait can depend on the value at risk in the transaction, and the likelihood of it unexpectedly failing. In practice, waiting long enough will make that transaction an immutable part of the earlier history of the Bitcoin blockchain. In contrast, in many private blockchains, committed transactions are more like normal database transactions and so, when accepted under the blockchain's consensus protocol, will immediately be a permanent part of the ledger.

Digital Assets

One of blockchain's most distinctive capabilities is allowing the creation and secure transfer of *digital assets*. Normally when you give information or digital files to someone, then you both end up with a copy. However, the fundamental characteristic of *property* is that of exclusion: when I have property, no one else has it; and when I transfer that property to you, then I no longer have it. Blockchain transactions and the globally visible blockchain ledger allow everyone to recognize and check the transfer of control or ownership of digital assets registered on the blockchain. This is how blockchain technology supports digital assets. The two most important kinds of digital assets have been discussed earlier: cryptocurrencies and tokens.

Cryptocurrencies are normally 'baked in' to the core platform of public blockchains. They have a kind of symbiotic relationship: the blockchain enables exclusive ownership and secure transfer of the cryptocurrency, and the cryptocurrency enables the incentive mechanism for the operation of the blockchain. Cryptocurrency uses cryptography to control the issuance of money (i.e. minting new coins) and to secure its transfer. Transfers are performed and recorded as financial transactions on a ledger. This virtual money can be transferred directly between users without using a trusted authority such as a bank. The first cryptocurrency, Bitcoin, created in 2009, is still the dominant one in terms of total market value at the time of writing in 2018. There are many cryptocurrencies, most of which are managed through the basic platform capabilities of specific blockchains, such as Ethereum's Ether, Ripple's XRP, and Nxt's NXT. Platforms such as Ripple and Nxt also provide native capabilities to define new cryptocurrencies.

In contrast to cryptocurrencies, tokens are usually not implemented directly in the core platform of a blockchain. Instead, they are implemented on top of blockchain platforms, using transaction data or smart contract features provided by the blockchain. Bitcoin allows developers to add 40 bytes of arbitrary data to a transaction, which is then permanently recorded on the blockchain. Thus, Bitcoin has been used for purposes such as representing digital assets (like document notarization) or physical assets (like diamonds). Bitcoin can also represent tokens using 'overlay networks', for example, using so-called colored coins, where a portion of Bitcoins is tainted to represent and manage real-world assets. Other overlay networks define a completely new transaction syntax, such as Omni and Counterparty. In Ethereum, tokens are usually implemented using smart contracts that maintain a register or table of ownership of tokens. Regardless, as a digital asset, there is much more variation among tokens than there is among cryptocurrencies. Tokens might represent fungible (interchangeable) commodities or might represent unique or serialized assets. Tokens might represent rights to use a service or might represent shares or voting rights in a company. Tokens are often implemented with features allowing their independent transfer or sale, but it is also possible to implement tokens that are not transferable or have other limitations on their transfer.

1.3.2 Blockchain as a Computational Infrastructure

Software components are the fundamental building blocks for software architecture, and blockchain can be a software component offering computational capabilities. As discussed earlier in Section 1.1.2, smart contracts allow us to execute small programs on the blockchain.

Ethereum views smart contracts as a first-class element. Smart contracts on Ethereum can express triggers, conditions, and business logic, to enable complex programmable behaviours. Smart contracts are used by components connected to a blockchain to reach agreements and solve common problems with minimal trust. A common simple example of a smart contract-enabled service is *escrow*, which can hold funds until the obligations defined in the smart contract have been fulfilled. As escrow holder, a smart contract's code has control over the assets held. Smart contracts can also be used to enable machine-to-machine communication for Internet of Things (IoT) applications.

One of the main kinds of architectural decisions is about which pieces of functionality should be allocated to which components. *For blockchain-based systems, this includes the key decisions about which parts of the data and computation should be placed on-chain or kept off-chain.* Parts of an application can be implemented inside the blockchain component using the blockchain ledger and smart contracts. However the amount of computational power, data storage space, and control of read accesses on a blockchain can be limited. So, parts of an application implemented outside the blockchain component might host off-chain data and application logic. Blockchain transactions and their effects sit at the interface between on-chain and off-chain functions.

A common practice is to store hashed data, metadata, and some small-sized public data on-chain and to keep large or private data off-chain. Due to the limited size of the data store provided by the blockchain, an off-chain data store is necessary for some applications. There are existing platforms providing a data layer on top of the blockchains, such as Factom,[5] which stores only the hash of the private data and small amounts of public data in their own blockchain. Distributed data storage, like IPFS,[6] or systems using DHTs (distributed hash tables) are also sometimes used in combination with blockchains to build decentralized applications.

Blockchain computation has a closed-world assumption; smart contracts can usually only examine state that is stored on the blockchain ledger. So in order to interact with the external world, *oracles* are invoked to bring external state into the blockchain. There are various sorts of oracles: some are like normal users of the blockchain and merely record facts about the world as normal transactions on the ledger; while others are components or nodes within the blockchain platform that can invoke smart contracts privileged to them. In either case, oracles typically become a trusted party for the respective data about the external world.

[5] https://www.factom.com/.

[6] InterPlanetary File System (IPFS)—https://ipfs.io/.

1.4 Blockchain Non-functional Properties

Besides blockchain's main functionality described above, software architects need to understand the non-functional properties of a system. We next give an overview of these for blockchain and touch on the implications for systems built on blockchain. As we will discuss in Section 1.5, understanding these issues is of central importance in the design of blockchain-based systems.

1.4.1 Non-functional Properties and Requirements

When specifying a system, software engineers often distinguish functional requirements from non-functional requirements. For a computer system, simple functional requirements characterize the relationship between observable inputs and outputs. Non-functional requirements (NFRs) are needs expressed for non-functional properties (NFPs), which are also known as 'qualities', or 'ilities'. These include characteristics such as cost, security (confidentiality, integrity, availability, privacy, non-repudiation), performance (latency, throughput), modifiability, and usability.

NFRs are expressed separately from functional requirements because they are often 'cross-cutting concerns' that span many system functions. For example, a requirement for the scalability of system performance might constrain the resources that can be used to respond to a given level of concurrent demand in a timely manner, up to some limit on that demand. The demand in this requirement would typically be a mix of many different kinds of system functions in normal usage.

Different use cases carry different NFRs. For example, in safety-critical industries such as medical devices or aerospace systems, NFRs for safety are paramount. In enterprise software systems, regulatory requirements often constrain NFPs such as privacy and data integrity. In regulated industries, legislation or regulation can provide constraints on minimum standards for critical NFPs within the industry. These constraints may be mandated to provide consumer protections or to manage systemic risks or negative economic externalities within the industry. NFPs are also important in understanding innovation: NFPs are quality or performance dimensions for technology, and technological progress pushes out the frontiers of performance on these various dimensions. Orders of magnitude improvements in performance on NFP dimensions open up possibilities for new markets and new business models using that technology innovation.

1.4.2 Non-functional Properties of Blockchain

Compared to conventional centralized databases and computational platforms (on-premise or cloud), blockchains can reduce some counterparty and operational risks

by providing neutral territory between organizations. Blockchain technologies may provide advantages for immutability, non-repudiation, integrity, transparency, and equal rights. If data is contained in a committed transaction, it will eventually become in practice *immutable*. The immutable chain of cryptographically signed historical transactions provides *non-repudiation* of the stored data. Cryptographic tools also support data *integrity*, the public access provides data *transparency*, and *equal rights* allows every participant the same ability to access and manipulate the blockchain. These rights can be weighted by the compute power or stake owned by the miner.

Trust in the blockchain is achieved from the interactions between nodes within the network. The participants of a blockchain network rely on the blockchain network itself rather than relying on trusted third-party organizations to facilitate transactions. These five properties (immutability, non-repudiation, integrity, transparency, and equal rights) are the main properties supported in existing blockchains.

Data privacy and *scalability* are two points of criticism of public blockchains. As discussed earlier, in this setting privacy is limited: there are no privileged users, and every participant can join the network to access all the information on blockchain and validate new transactions. Often, applications need to find an acceptable trade-off between data privacy/confidentiality and transparency.

Current public blockchains have scalability limits on:

1. the size of the data on blockchain, due to the global replication of all data across all full nodes.
2. the transaction processing rate. For example, mainstream public blockchains can only handle on average 3–20 transactions per second,[7] whereas mainstream payment services, like VISA, handle an average of 1700 transactions per second.[8]
3. the latency of data transmission. Because nodes can have a local copy of the blockchain, read latency can be good, but because updates must be propagated across a global network, write latency is typically not good. The number of transactions included in each block is also limited by the bandwidth of nodes participating in leader election (for Bitcoin the current bandwidth per block is 1 MB). Latency between submission and confirmation that a transaction has been included on a blockchain is affected by the consensus protocol. This is around 1 h (10-min block interval with 6-block confirmation) on Bitcoin and around 3 min (14-s block interval with 12-block confirmation) on Ethereum.

[7] https://blog.ethereum.org/2016/01/15/privacy-on-the-blockchain/.

[8] https://usa.visa.com/run-your-business/small-business-tools/retail.html.

1.5 Blockchain Architecture Design

Understanding the main functional and non-functional properties of blockchain described above is vital for good architectural design for systems using Blockchain, which we introduce below.

1.5.1 Software Architecture: Design and Analysis

The software architecture of a software-based system is the high-level structure of relationships between software elements (components and connectors) in the system. In the creation of a software architecture, there are many possible options for these structures, and the choices between these options are important design decisions. A key realization in the discipline of software architecture is that these design decisions have a critical impact on a system's ability to meet NFRs. Given a design candidate for a software system, software engineers may use qualitative, analytical, or simulation-based tools to evaluate the design for its predicted ability to achieve an NFR.

To achieve an NFR, the right design decisions must be made, and each design decision will impact a number of NFPs, either positively or negatively. Often this will lead to conflicts between NFPs, so it is important to manage trade-offs between these when designing a system. An important part of software architecture as a practice is to document the design for a system, including the rationale for why specific design options were chosen.

1.5.2 Designing Blockchain-Based Applications

Blockchain-based systems can be different from traditional systems in various ways, as outlined below. Chapter 3 provides a more comprehensive taxonomy.

Admittance of Processing Nodes In a public blockchain system, such as Bitcoin, anyone may become a processing node (or 'miner'). In private blockchain systems, the admittance of processing nodes is controlled by its governing bodies. Public blockchains provide very low barriers to entry for new participants, which can facilitate competition, innovation, and productivity. However, public blockchains typically do not mandate authentication of those participants, which creates challenges regarding AML/CTF and tax avoidance. Private blockchains can impose more controls on authentication and access, which can partly address those regulatory concerns.

Consensus Mechanism Most public blockchains use *Nakamoto consensus*, where processing nodes by convention treat the longest history of blocks as the author-

itative history. The rate at which blocks can be created is limited, often by using a proof-of-work mechanism, whereby a processing node can only add a new block by demonstrating that a difficult task has been completed. Proof-of-work is widely used, but the auxiliary effort required to complete the difficult task can be economically inefficient. In a proof-of-stake system, the processing node that can add a new block in the next round is determined by the size of its stakeholding in the global blockchain and/or in that round. Proof-of-stake can be more efficient than proof-of-work but to date has been less widely used. Proof-of-work implementations have demonstrated operational stability over years. Other consensus mechanisms have been proposed. On private blockchains where there are a smaller number of more trustworthy processing nodes, conventional replication algorithms such as Practical Byzantine Fault Tolerance (PBFT) can be used instead of Nakamoto consensus.

Representation of Transactions A distributed ledger may record financial transactions, such as in Bitcoin. However as a shared database, a distributed ledger might allow other kinds of data to be recorded. In particular, the data recorded for a transaction may be the text of a computer program, and the integrity check for that transaction may involve executing that program. This allows participants to create smart contracts, which allow transactions to represent behaviour as well as data.

There are several kinds of blockchains, and to provide more general insights we take a broad view. For example, the Bitcoin system is a public blockchain, which allows unfettered public participation in both its operation and use. Other well-known systems, such as the Ethereum[9] blockchain, are similar in this regard. It is possible to use a separate instantiation of the Bitcoin or Ethereum computer programs to operate a blockchain within a private context, for example, on a virtual private network. These would then be one kind of 'private blockchain'. Note that operators of such networks would not normally use proof-of-work consensus in private networks, because of limitations with that kind of consensus and because other controls or assumptions can be used to address integrity. The access controls possible for private networks and private computer systems allow for greater administrative control over such blockchains. However, the software for public blockchains is not always the best technical solution to use in a private setting. Many industry consortia, such as Hyperledger,[10] R3,[11] and Ripple,[12] are actively developing specialized private blockchain solutions. These typically support a smaller number of processing nodes than public blockchain solutions, but can provide confidentiality and increased performance.

Recently, proof-of-authority (PoA) has gained popularity as a consensus mechanism for private or permissioned blockchain systems, and implementations in

[9] https://www.ethereum.org/.

[10] https://www.hyperledger.org/.

[11] https://www.r3.com/.

[12] https://ripple.com/.

	Permission-less	Permissioned
Public	**Consensus**: Proof-of-X **Permission management** • Blockchain layer • Application layer (optional) **Incentive**: Blockchain layer	**Consensus** • Proof-of-X • PBFT, Federated consensus, Round Robin *etc.* **Permission management** • Blockchain layer • Application layer (optional) **Incentive**: • Blockchain layer • Governance around permissions
Private	**Consensus** • Proof-of-X • PBFT, Federated consensus, Round Robin *etc.* **Permission management:** • Blockchain layer • Network layer • Application layer (optional) **Incentive**: Governance around access	**Consensus** • Proof-of-X • PBFT , Federated consensus, Round Robin *etc.* **Permission management:** • Blockchain layer • Network layer • Application layer (optional) **Incentive**: Governance around access permissions

Fig. 1.4 Core components of different types of blockchain

	Permission-less			Permissioned	
Public	Immutability	+++	(#Nodes, Consensus, Topology)	Immutability	++
	Integrity	+++	(#Nodes, Consensus, Topology)	Integrity	++
	Transparency	++	(Access control)	Transparency	++
	Availability	+++	(#Nodes, Topology)	Availability	++
	Performance	+	(Consensus, latency)	Performance	++
	Cost Efficiency	+		Cost Efficiency	++
Private	Immutability	+		Immutability	+
	Integrity	+		Integrity	+
	Transparency	+		Transparency	+
	Availability	+		Availability	+
	Performance	+++		Performance	+++
	Cost Efficiency	+++		Cost Efficiency	+++

Fig. 1.5 Non-functional properties of different types of blockchain

Ethereum client software are gaining adoption. PoA assigns the right to mine new blocks to a set of authorities (blockchain accounts, i.e. key pairs) that produce new blocks.

Figures 1.4 and 1.5 show the core components of different types of blockchains and the corresponding quality impacts. In practice, the lack of standard and reliable technology evaluation criteria makes a precise comparison difficult. When building applications based on blockchains, we need to systematically consider the features and configurations of blockchains and assess their impact on quality attributes for the overall systems. For example, a blockchain transaction is not appropriate for all data: because it is replicated globally, transactions should not contain very large data nor plain-text data which must be kept confidential. Similarly, for competitors within an industry consortium, private blockchains may not be private enough to provide

normal levels of commercial confidentiality for business operations, competitive position, and customer relationships. Consequently there are choices about what data should be stored *on-chain* inside transactions and what should be stored *off chain*, in external systems. Although a specific blockchain platform may have significant limitations, if it can be combined in a design with other components in an effective way, then many kinds of business challenges can be targeted by blockchain-based systems.

1.6 Summary

Blockchains and distributed ledgers are currently very hot topics in computing. In this chapter, we introduced what they are and why there is wide interest in them within various application areas. To provide clarity, we have defined the most important terms used in this book.

Then we discussed, at a high level, the most important aspects for the software architect and engineer aiming to develop a blockchain-based application: what does blockchain offer in terms of functional and non-functional properties and how to approach designing blockchain-based applications?

In the next chapter, we will present an in-depth view of some existing blockchain platforms. Chapter 3 then discusses the conceptual differences between various blockchain technologies and their implications for architectural design. Concrete use cases are discussed in Chapter 4.

1.7 Further Reading

This chapter is partly based on our earlier works (Staples et al. 2017).

The original conception of blockchain was first discussed in the Bitcoin paper (Nakamoto 2008). A more complete introduction of blockchain and Bitcoin can be found in Swan (2015) and Antonopoulos (2015). The original conception of smart contracts predated blockchain technology (Szabo 1997). Smart contracts were originally a way of realizing legal contracts in physical computing systems. However, in the blockchain context, smart contracts are not necessarily related to legal contracts.

Comprehensive surveys on the state of the art of existing cryptocurrencies include Morisse (2015), Bonneau et al. (2015), and Tschorsch and Scheuermann (2016). The market value of cryptocurrencies can be found on http://coinmarketcap.com.

Some government reports discuss potential applications of blockchain in various scenarios, for example, Walport (2016) and Staples et al. (2017). The case of the UN refugee camp's use of blockchain has been described by Juskalian (2018). The experiments on blockchain for programmable money to automate disability support grants are described by Royal et al. (2018).

For an interesting historical account of a community dollar, read *The Island of Stone Money* by Friedman (1991). It describes the island Yap in Micronesia, where currency was held by ownership designation on big stones.

The software architecture of a software-based system is the high-level structure of relationships between software elements (components and connectors) in the system (Clements et al. 2003; Bass et al. 2012). The design of software architecture needs to consider non-functional requirements, which are needs expressed for non-functional properties. These include characteristics such as cost, security and dependability (Anderson 2008; Avizienis et al. 2014) (confidentiality, integrity, availability, maintainability, safety, reliability, privacy, non-repudiation), performance (latency, throughput), modifiability, and usability.

Chapter 2
Existing Blockchain Platforms

This chapter introduces some of the most prominent and representative blockchain platforms, including Bitcoin, Ethereum, and Hyperledger Fabric. Other blockchain platforms are also briefly discussed.

2.1 Bitcoin

Bitcoin is a cryptocurrency operated on a peer-to-peer (blockchain) network. Unlike traditional banking and payment systems, Bitcoin is based on decentralized trust; there is no central trusted authority in the Bitcoin system. Trust emerges from the interactions of different participants in the ecosystem. Figure 2.1 gives an overview of Bitcoin system. In the Bitcoin system, there is a distributed ledger that stores all Bitcoin transactions. The content of the ledger is replicated across many geographically-distributed processing nodes within the Bitcoin network. We described its operating principles in the sidebar on page 4 in an informal, non-technical way.

For clarity, we will refer to the tokens on the Bitcoin blockchain using their currency code *BTC*. There are three main types of nodes within the Bitcoin network. (1) Users with wallets: a wallet maintains the key pairs of the user, which are used to authenticate the transactions initiated by the user by means of digital signatures. (2) Miners that compete with each other to add new blocks to the shared ledger as the authoritative source of all the transactions. (3) Exchanges, i.e. places where users can buy BTC in exchange for other currencies.

Below we describe the pieces and concepts out of which the Bitcoin blockchain is built. Many of these concepts were known before Bitcoin, but their combination as a blockchain was new and has created a technology with interesting properties. Many of these concepts are also used by other blockchain platforms and distributed ledger technologies. Interestingly, it is not only the technical concepts that make

X. Xu et al., *Architecture for Blockchain Applications*,
https://doi.org/10.1007/978-3-030-03035-3_2

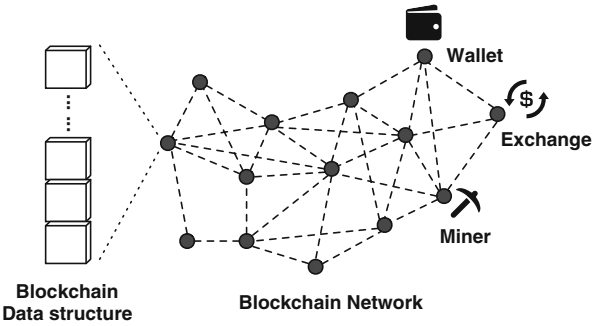


Fig. 2.1 Overview of Bitcoin system

public blockchains work but also the economics and social incentives. For instance, network participants who have invested in the most (computational) power are also rewarded the most in the Bitcoin system, and because of that investment would be penalized the most if trust in the network eroded causing the value of BTC to decrease. So, they are incentivized to act in the interest of the network.

2.1.1 Bitcoin Transactions

A simplified life cycle of a blockchain transaction was introduced in Fig. 1.3.[1] A peculiarity of Bitcoin is the way transactions are linked: they transfer currency from a number of source addresses to a number of destination addresses. As shown in Fig. 2.2, the outputs of transactions become the inputs of new transactions. If the sum of the outputs is less than the sum of the inputs, the difference is interpreted as an additional output that serves as a fee to the miner who creates the block containing this transaction. The transaction fee is an incentive for miners to contribute their computing power. As a result, miners tend to optimize block creation by preferring transactions with higher fees. The transaction fee is often the only variable that client software asks Bitcoin users to choose consciously when creating a new transaction.

However, transactions can experience delay due to other factors. One important factor is that transactions must arrive (roughly) in-order, for a node (and the network) to be able to process them fast. Incoming transactions are handled by the so-called *mempool*. If the referenced input transactions, called *parents*, are as-yet unknown, a miner will delay the inclusion of the new transaction—it is then a so-called *orphan*. Miners may choose to keep orphans in the *mempool* while waiting

[1]The life cycle is simplified in that transactions may not always make it to the 'confirmed' state after the initial validation, e.g. when a conflicting transaction is included instead.

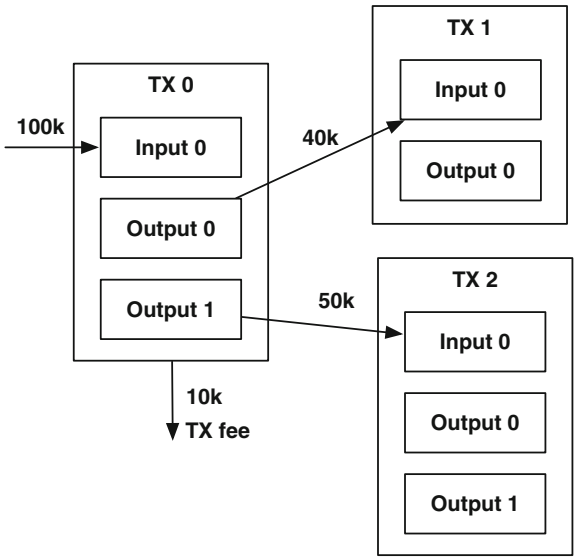


Fig. 2.2 Bitcoin transactions

for the parent transactions to arrive, but they may also expunge orphans after a timeout they choose. A second factor that could come into play, albeit one that only experienced users will set, is so-called *locktimes*: a transaction can contain a parameter declaring it invalid until the block with a certain sequence number has been mined.

2.1.2 Script

Bitcoin uses a scripting system for transactions. The script language is called *Script*, which is simple, stack-based, and processed from left to right. Script is not Turing complete. It has limited complexity without looping and complex flow control. A script is a list of instructions associated with each transaction that describes how the BTC transferred with the transaction can be spent. A *locking script* is placed on an output, which specifies the conditions that must be met to spend the BTC. An *unlocking script* is placed in an input that 'solves' or satisfies the conditions of the locking script. To validate a transaction, the unlocking script and the locking script are combined and executed. If the result is *true*, the transaction is valid. The most common case implements a simple transfer, referred to as Pay-to-PubKey-Hash (P2PKH), where the locking script specifies which (hashed) public key and corresponding signature are required to unlock the output. In other words: only the holder of the designated key pair can spend the output.

Script provides certain flexibility to change the parameters of the conditions to spend the transferred BTC. For example, a transaction can require multiple keys

and signatures. *OP_RETURN* is a Script keyword, called *opcode*, used to mark a transaction output as invalid. *OP_RETURN* has been used as a standard way to embed arbitrary data to the Bitcoin blockchain for other purposes, like representing assets. By design, Script programs are pure functions, which cannot poll external servers or import any external state. An *oracle* can be used to include external state into the blockchain execution environment. See Section 5.4.2 for more details.

2.1.3 Mining

Mining nodes compete in a *proof-of-work* system to create new blocks by solving hard cryptographic puzzles. Bitcoin uses the *hashcash*[2] proof-of-work function. Some miners are full nodes, maintaining a full copy of the blockchain data structure, while others are lightweight nodes participating in pool mining and depend on a coordinating pool server to maintain a full replica.

Miners are always listening for new transactions and new blocks, as do all the nodes. When a transaction reaches a mining node, it is verified, included into the *mempool*, and propagated to the network. To the miners, the arrival of a new block means the completion of the previous round of competition and an announcement of a winner. The end of one round of a competition is the beginning of the next round. To start mining a new block, the miner first removes the transactions from the *mempool* that belong to the received block and aggregates a set of the remaining valid transactions into a candidate block, reassessing the validity of each transaction at the point where it is added to the candidate block. It also adds the so-called *coinbase* transaction as the first transaction to the list of transactions for the new block. The coinbase transaction pays a block reward to the miner, which is another incentive for mining (in addition to the transaction fees). Then the miner constructs the block header, which includes a hash of the previous block and a summary of all the transactions in a binary tree, called a *Merkle tree*, for more efficient searching.

Next, a solution to the proof-of-work function needs to be found. It requires finding a value for a free field in the block header, the nonce, which leads to the block hash being smaller than a given threshold. In short, finding such a nonce requires a lot of trial and error: at the time of writing, on average 2.4×10^{21} nonces are tried and hashes computed per Bitcoin block, but across the global network without coordination on which nonces to try, and therefore highly redundant. The threshold is adjusted over time to ensure that the average time between blocks is around 10 min. In other words, the puzzle is so hard that all Bitcoin miners around the world together still take 10 min on average to solve it. Every candidate block is a new puzzle, and the likelihood to solve it first is proportional to the compute power invested relative to all compute power in the network.

Once a solution is found, the result is inserted into the block header, and the new block is immediately propagated to the network. This situation is depicted in

[2] https://en.bitcoin.it/wiki/Hashcash.

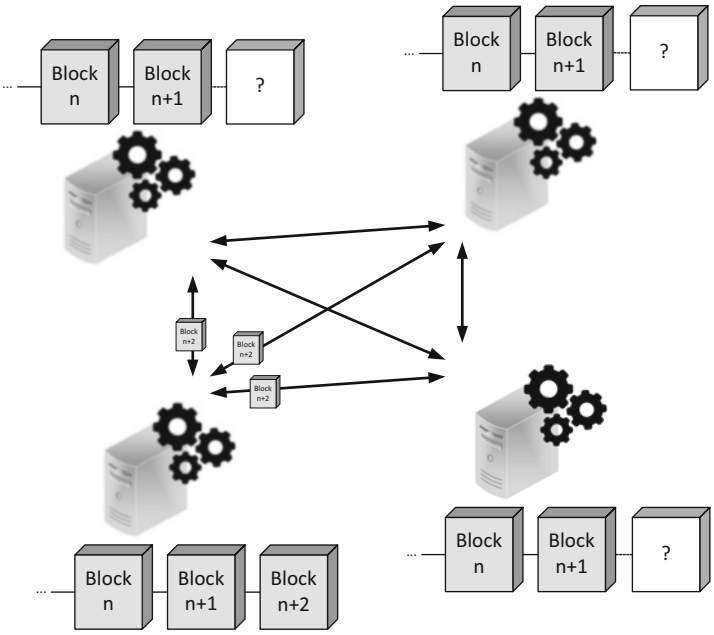


Fig. 2.3 A new block: the miner in the lower left corner found the next block $n+2$ and broadcasts it to the network

Fig. 2.3. The nodes receiving the new block verify it, then include it into their replica of the blockchain data structure, as shown in Fig. 2.4, before starting the search for the next block.

Mining is also the way in which new coins are minted: the coinbase transaction has an output but does not consume any inputs. Therefore it creates new BTC. At the time of writing, each coinbase can have an output of 12.5 BTC, paid to the miner who created the block.

2.1.4 Accounts and State

An account in Bitcoin is associated with a cryptographic key pair. The public key is used to create the account address, which is somewhat similar to the bank identifier and account number in traditional banking (or their combination as an International Bank Account Number, IBAN). BTC can be sent to an account address. The corresponding private key is required to sign transactions originating from the account. Because the source account is known, every node in the network can verify the signature. This is achieved with the locking/unlocking scripts mentioned above.

The state of the blockchain, and specifically the *account balances* of all users, results from the set of transactions and the *genesis block*, which is the first block

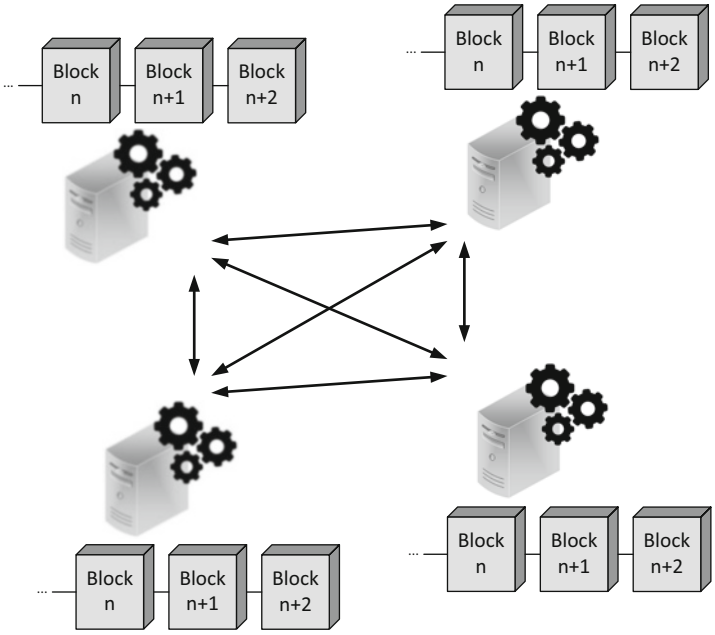
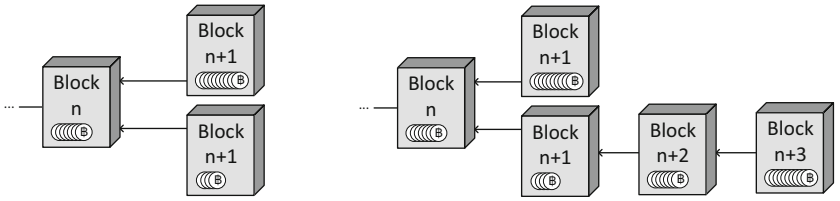


Fig. 2.4 After the new block has been propagated (from Fig. 2.3), the other nodes in the network accept it and append it to their local copy of the blockchain data structure

(block number 0). Some accounts might be preloaded with an initial account balance from the beginning, i.e. in the genesis block. When a transaction from *A* to *B* occurs, *A*'s balance is reduced by that amount, and *B*'s account is increased by that amount. The miner *C* may also receive a transaction fee, if *A* specified that, in which case *B* receives less than *A* sends. The transaction becomes part of the ledger when the miner creates a block that includes it and when that block is included by consensus in the blockchain data structure. Then the transfer has occurred. The miner *C* is paid a block reward for this new block through the coinbase transaction mentioned above.

Bitcoin does not track account balances explicitly. The Bitcoin blockchain platform has exactly two first-class elements: transactions and blocks. The account balance is therefore derived as the sum of *unspent transaction outputs* (abbreviated to *UTXO*) that an account has control over. Bitcoin's record-keeping model is therefore referred to as UTXO, in contrast to Ethereum's account/balance model. Either way, every node has access to the full transaction history and thereby knows which account holds how much currency. Because accounts are pseudonymous, typically the persons holding each account are not known to most actors. As transactions are grouped into blocks, the entire system moves from one discrete state to another through the addition of whole blocks each containing many transactions.



(a) Fork in the blockchain (b) Fork decided, the longer chain wins

Fig. 2.5 A fork in the blockchain offers two possible versions of the new state (**a**), which are decided by the additional blocks $n + 2$ and $n + 3$ (**b**)

2.1.5 Nakamoto Consensus

Most public blockchains use Nakamoto consensus, which was introduced with the Bitcoin blockchain. In Nakamoto consensus, processing nodes by convention treat the longest history of blocks as the authoritative history—it is called the *main chain*. Before one chain is longer than the other, it is unclear which state will prevail. This situation is illustrated in Fig. 2.5a and resolved in Fig. 2.5b. In combination with proof-of-work, the longest chain corresponds to the one that (on average) received most computation.

Mining the next block is a constant global race between ten thousands of computers in the Bitcoin network. Multiple computers might more or less simultaneously find and announce the next block, say $n + 1$ in the example above. The decision which version of block $n+1$ becomes part of the main chain is made by the winning block $n + 2$ and to which block $n + 1$ it refers as predecessor. However, there might be multiple conflicting versions of block $n + 2$ referring to different predecessors $n+1$. While possible, the Bitcoin protocol renders it very unlikely that such parallel forks continue for more than a block or two (unless the network is separated, which is unlikely for larger portions of the Internet).

Due to this possibility, users want to determine with high probability that a transaction is permanently included in the blockchain. Users therefore wait for several blocks to be mined after the first inclusion of their transaction to gain confidence that the block including their transaction is part of the main chain. Each of these subsequent blocks is called a *confirmation block*, and when sufficiently many confirmations occurred after the transaction block inclusion, then the transaction is considered *committed*. Depending on the importance of the transaction and the risk of it being excluded, the number of required confirmation blocks might need to be higher or lower. A default number is six blocks (one for inclusion and five confirmation blocks), though the source of this number is somewhat arbitrary.[3] This equates to a probabilistic guarantee meeting a (usage-specific) likelihood threshold.

[3]This number goes back to an early wallet of Bitcoin: its UI suggested that the transaction was finalized after six blocks. Commit after six blocks also corresponds to a double-spending attack

2.1.6 Deflationary Cryptocurrency

Blockchains that support primarily a cryptocurrency, like Bitcoin, are regarded as the first generation of blockchains. Bitcoin provides a deflationary cryptocurrency by defining certain rules. The total amount of BTC that will be released over the life cycle of Bitcoin is 21 million. As discussed earlier, new BTCs are issued during the mining process. Each time a new block is mined and successfully added into the blockchain, new BTCs are rewarded to the miner who created the valid block. The reward is halving every 210,000 blocks. Initially, the reward was set to 50 BTC and fell to 25 BTC in late 2012. Mining rewards in Bitcoin will run out in 2140, when no more new BTC will be issued (unless the rules change).

2.1.7 Wallets

A software wallet allows users to manage a collection of private keys corresponding to their accounts and to create and sign transactions on the Bitcoin network. A wallet may include a full node but does not have to. SPV (Simplified Payment Verification)[4] nodes maintain only part of the blockchain and verify if and when particular transactions are included in a block without downloading the entire blockchain ledger. That allows running wallets on resource-constrained devices, such as smart phones.

Hardware wallets are specialized devices that provide part of the above functionality, typically in combination with suitable software. A common split is to create and store private keys on the hardware; they never leave the device. Public keys are exported, so that payments can be received. For outgoing payments, the unsigned transaction is sent from the software to the device, verified by the user on the display of the device and confirmed with a PIN, and then signed by the device and sent back to the software wallet.

To avoid accidental loss of private keys, and thereby loss of the ability to spend one's funds, there are *cold-storage solutions* as backups. These work by storing a representation of the keys in a way that is independent of the user's current hardware wallets and machines. The simplest way is to write key pairs down on paper. More user-friendly methods work by writing (or printing) 12 or 24 words out of a dictionary on paper. All cold-storage solutions of course need to be protected from conventional threats. An interesting alternative to paper is custom metal plates in which the keys are set and physically locked in place—these have the advantage of being fireproof.

with 10% of the mining power having a 0.1% chance of success, as outlined in a theoretical analysis paper.

[4] https://bitcoin.org/en/glossary/simplified-payment-verification.

2.1.8 Exchanges

Bitcoin exchanges are places (usually websites) to buy Bitcoin in exchange for other currencies (fiat currencies like US$ or cryptocurrencies). During this process, the exchange holds currency on behalf of users, which makes exchanges a kind of trusted party within the Bitcoin system. Clients may choose to ask the exchange to transfer purchased Bitcoin to an address under their control. But until they do that, if the exchange's system fails, their users may lose control of 'their' Bitcoin. Exchange markets provide liquidity for cryptocurrency, which supports its real-world value and thus underpins the incentive mechanisms at work for miners to operate the Bitcoin blockchain. Therefore, exchanges are key stakeholders for public blockchain platforms.

2.2 Ethereum

Bitcoin led the development of the first generation of blockchain systems, providing a public ledger to record cryptographically signed financial transactions. Bitcoin has limited support for programmable transactions, and only very small pieces of auxiliary data can be embedded in the transactions to serve other purposes. The second generation of blockchain systems provides a general-purpose programmable infrastructure where the public ledger not only stores financial transactions but also has facilities to deploy and execute programs on the blockchain system. The Ethereum blockchain platform views smart contract as a first-class element and includes a virtual machine for executing smart contracts.

2.2.1 Ethereum Protocol

Ethereum is configured to have a relatively short time interval between blocks: 13–15 s on average. This of course addresses the issue of long delays of Bitcoin transactions, where the inter-block time averages around 10 min. Ethereum's inter-block time is not many times longer than the time required to propagate information throughout the global blockchain network. Because of that, it is much more likely in Ethereum that multiple new (competing) blocks are created concurrently at similar times. A *stale block* is one that was successfully created by a miner, propagated to the network and verified by some nodes as being correct, but is eventually discarded when another longer chain achieves dominance. As shown in Fig. 2.6, a stale block is created when miner B and miner C find new blocks and propagate their blocks almost at the same time. In Bitcoin, the probability of finding a block at the same time is relatively low because the average inter-block interval is 10 min.

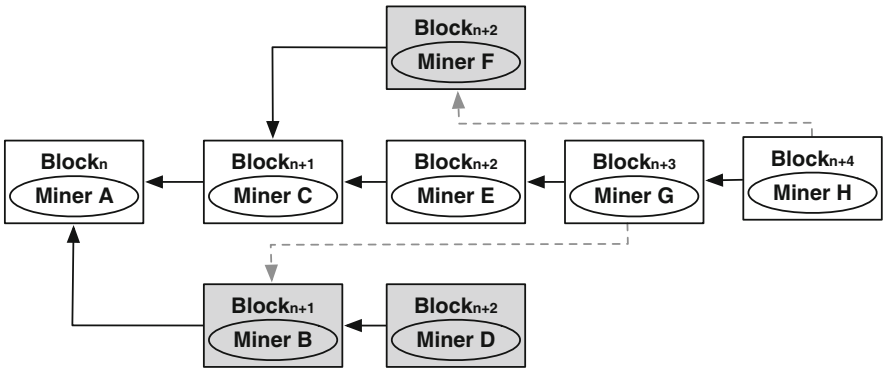


Fig. 2.6 Ethereum blockchain

The Ethereum blockchain uses a modified GHOST (Greedy Heaviest Observed Subtree) protocol, which was proposed as a way to address this problem. In the GHOST protocol, miners reference competing independently mined blocks (so-called *uncles* in Ethereum terminology; in the figure, $\text{Block}_{\text{MinerB}}$ and $\text{Block}_{\text{MinerF}}$ are uncles), to add weight to their chain in the calculation of which chain is longest or has the highest cumulative difficulty. In Ethereum, not the longest chain wins, but the 'heaviest'—and recognized uncles contribute to the weight. This recognition of concurrent work allows shorter inter-block times which can improve throughput. The recognition is backed by a strong financial incentive: miners of uncle blocks receive 87.5% of a standard block reward. For every uncle included in the block, the miner gains an additional 3.125% and increases the weight of the chain including its block.

2.2.2 Ethereum Transactions

A high-level life cycle of a transaction is discussed in Fig. 1.3. Here we discuss the life cycle of an Ethereum transaction, from it arriving in the transaction pool until it is committed. As shown in Fig. 2.7, the transaction life cycle can be split into consecutive phases: (i) the announcement of the transaction in the system; (ii) the inclusion of the transaction in a newly mined block on some branch of the chain; (iii) the block in which the transaction is included is part of the main chain; and (iv) the commitment of the transaction after sufficiently many blocks are subsequently mined.

Before a transaction is included in a block, it gets validated. This includes checks of the digital signature, parameters such as the nonce (sequence number of transactions relative to a given source account), and that there are sufficient funds in the source account.

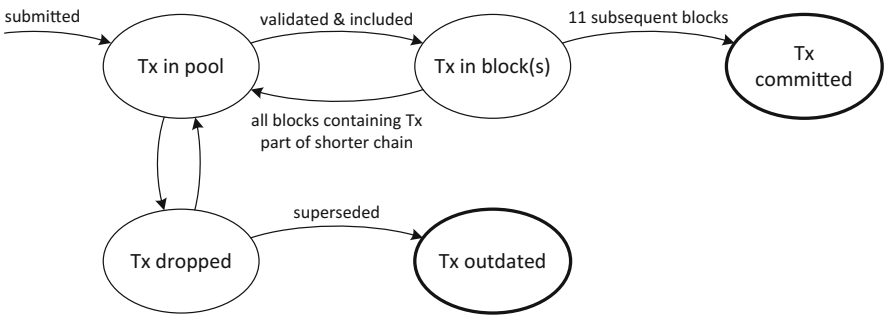


Fig. 2.7 Life cycle of an individual Ethereum transaction *Tx* (notation: state machine). © 2017 IEEE. Reprinted, with permission, from Weber et al. (2017)

Note that Step (ii) above is not sufficient to guarantee that a transaction is permanently added to the blockchain: if the blockchain forks, then the block comprising the transaction may simply be discarded, and it could be re-included later. While uncle blocks may be recognized in Ethereum, their content is discarded at any rate. If all blocks that include the transaction become part of a shorter chain than the main chain (i.e. they are *uncles*), then the transaction returns to the transaction pool. This might happen repeatedly. While the transaction is in the pool at a miner, it may also be dropped at the discretion of the miner. It is impossible for any node in the network to know with certainty whether all miners have dropped the transaction. Only when the nonce of the transaction becomes outdated, i.e. another transaction from the same source account with the same nonce has been committed, can a node be certain that the old transaction cannot be included in any valid block. Otherwise the transaction might later resurface and be included in the chain.

Ethereum uses proof of work, like Bitcoin, and the GHOST protocol states that the longest/heaviest chain becomes the main chain. Therefore, like for Bitcoin, Ethereum users wait for X confirmation blocks before seeing a transaction as committed. Due to the higher rate of uncle blocks, X is typically higher than for Bitcoin: 12 blocks (block that includes the transaction + 11 confirmation blocks) are typical on Ethereum. For an in-depth discussion, see Section 11.6.2.

2.2.3 Smart Contract

Smart contracts are programs deployed and run on a blockchain system. Smart contracts can express triggers, conditions, and business logic to enable complex programmable transactions. On the Ethereum blockchain, smart contract developers

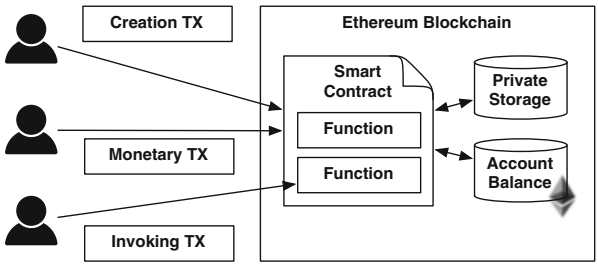


Fig. 2.8 Ethereum smart contract

can use high-level programming languages, like Solidity,[5] to define smart contracts. Solidity code is compiled into a low-level stack-based bytecode language, which is run by the Ethereum Virtual Machine (EVM) included in every node within the Ethereum blockchain network. To guarantee coherence across different copies of the blockchain, EVM code is specified to execute deterministically. Smart contracts in Ethereum should not be seen as representations of legal contracts that should be 'fulfilled' or 'complied with'; rather, they are more like agents that can be invoked within the Ethereum execution environment.

As shown in Fig. 2.8, a smart contract is deployed on the blockchain through a contract creation transaction. The data payload of the transaction contains the object code of the smart contract. The signature of the transaction sender authorizes the transaction to create the smart contract on the blockchain. After the contract creation transaction is successfully included in to the blockchain, the smart contract is identified by a contract address. Every smart contract has a blockchain account which can hold Ether (the Ethereum cryptocurrency) and internal state. Thus, an Ethereum smart contract account contains:

- A piece of executable code
- An internal storage to store its internal state
- An amount of Ether, i.e. the contract balance

After a smart contract is successfully deployed on blockchain, blockchain users can transfer Ether to the smart contract by using a basic monetary transaction.

Smart contracts are programs that need to be externally invoked. Blockchain users can invoke the functions defined in the smart contract by sending contract-invoking transaction to the address of the smart contract. The contract-invoking transaction contains (1) the interface of the function being invoked and its parameters in the data payload and (2) an amount of Ether to pay for the execution of the invoked functions. The signature of the transaction sender authorizes the data payload of the transaction to execute a smart contract. The functions defined in a smart contract can also be invoked by other smart contracts.

[5] https://solidity.readthedocs.io/.

2.2.4 Paying Fees in 'Gas'

A smart contract on the Ethereum blockchain is locally executed by every miner, and so consumes their computational resources. In a Turing complete language, it is not always possible to predict the computational resources that will be required by a program or even whether the program will terminate. It is important that the replicated execution of a nonterminating program does not freeze the whole network.

To limit the use of resources, and to compensate miners for the use of their computational resources, Ethereum uses the concept of *gas*, as a fee proportional to the required data storage and computation. In rough terms, there is a fixed gas cost for each transaction, plus variable gas cost for data (proportional to its size) and execution of a smart contract method (charged per bytecode instruction). There is an additional gas cost for the deployment of new contracts. The Ethereum yellow paper defines a detailed cost model. All costs in Ethereum follow a pricing table, specified in the unit *gas*. Gas cost is converted to Ether according to a user-defined *gas price*, i.e. how much Ether-per-gas the creator of a transaction is willing to pay. By default, Ethereum clients set the gas price to a market rate, an average over previously included transactions. The gas price can be set to 0, meaning the transaction sender is not offering a fee. Intuitively, users set higher gas prices if inclusion of their transaction is urgent for them and lower gas prices if inclusion can take time or may fail altogether—but this intuition does not always match reality as we discuss in Chapter 11.

Other than gas price, when users send contract-invoking transactions, a *gasLimit* must be set, which bounds the computation for a smart contract. The miner who successfully includes the transaction in the blockchain receives a transaction fee corresponding to the amount of gas the execution has actually used, multiplied by the gas price. An execution which requires more gas than *gasLimit* causes an exception, and the state of the smart contract is rolled back to the state before the execution.

To prevent denial-of-service attacks, Ethereum also defines a gas limit at the block level: the sum of gas used by the transactions included in a block cannot exceed this limit. The block gas limit is influenced by the miners, where each miner winning a block can vote to increase, decrease, or keep the current block gas limit. However, the block gas limit also limits the complexity for blocks which also bounds throughput.

2.2.5 Decentralized Application (dapp)

With Ethereum and its smart contract capabilities, the idea of decentralized applications (dapps) has gained popularity. A dapp is an application whose core logic resides in smart contracts and where the code is accessible to the users (typically as open-source). Therefore, the users do not have to trust any single entity: they

can inspect the code to understand what it does; and because it is run on top of a blockchain system and is deterministic, they can trust in its faithful execution.

The backend of a dapp is executed in a decentralized environment. This is different from the backend of normal apps which are executed on a centralized server. A dapp, like a normal app, can have frontend code and user interfaces that interact with its backend through an API. The frontend can be hosted as a website on a centralized server. A dapp could also, like a normal app, use decentralized data storage such as IPFS.[6] *State of the dapps*[7] is a directory of dapps running on Ethereum. This directory is also recorded on the Ethereum blockchain.

2.3 Hyperledger Fabric

Hyperledger is an umbrella project of open-source blockchains and related tools.[8] It is a global collaboration, hosted by the Linux Foundation since December 2015. Members are from domains such as finance, banking, Internet of Things, supply chain, manufacturing, and technology. There are currently more than 185 members and 8 ongoing projects, including Hyperledger Fabric. Hyperledger Fabric is a business blockchain framework, intended as a foundation for developing blockchain-based applications with a modular architecture. Data can be stored in multiple formats, and various consensus algorithms can be configured. Figure 2.9 gives an overview of Hyperledger Fabric system. More details can be found in the Hyperledger documentation.[9]

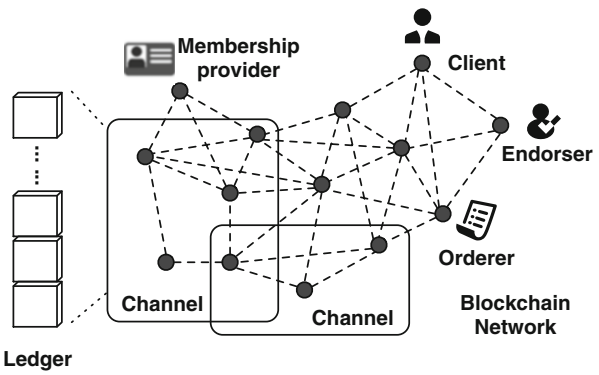


Fig. 2.9 Overview of Hyperledger Fabric system

[6] https://ipfs.io/.

[7] https://www.stateofthedapps.com/.

[8] https://www.hyperledger.org.

[9] https://hyperledger-fabric.readthedocs.io.

2.3.1 Permissioned Blockchain

Hyperledger Fabric is a private and permissioned blockchain. Members of a network need to enrol through a trusted membership service provider (MSP). All the participants of a Fabric network have known identities. Public keys are used as cryptographic certificates tied to organizations, network components, and end users. Data access control is applied on network and channel levels.

The concept of *channels* helps to address scenarios where privacy and confidentiality are important and reduced transparency is acceptable. A channel allows a group of participants to create a separate ledger of transactions, shared only with a set of members for that channel. A channel might cover the entire blockchain network, similar to a public blockchain system, or might include only a few participants from the entire network.

Channels are important for systems where participants might be competitors and do not want to disclose all of their transactions to each other. For example, a company will not want to disclose the identity of its customers or the volume of its sales to its competitors. If a company forms a channel with one of its customers, then only those two participants and no others can see the transactions on the associated ledger for that channel.

The ledger associated with a channel comprises two components: the world state and the transaction log. The world state is the latest state of the contents of the ledger. The transaction log records all historical transactions which have resulted in the world state. The ledger of a channel also contains a configuration block that defines information such as policies and access control lists.

2.3.2 Chaincode as Smart Contract

Hyperledger Fabric leverages container technology to host smart contracts called *chaincode* that comprise the application logic of the system. Chaincode can be implemented in programming languages such as Go or Java and is invoked through a transaction proposal. The execution of chaincode is based on the world state stored in the ledger for a channel.

2.3.3 Nodes

There are three types of nodes within a Hyperledger Fabric system: client, peer, and orderer.

- **Client:** A client acts on behalf of an end user. It connects to peers to communicate with the blockchain. A client node can create transactions and broadcasts messages to **Orderers** (see below) through *communication channels*.

- **Peer:** A peer node receives ordered state updates in the form of transactions from the orderers, commits transactions, and maintains the state of the ledger. Some peers can take a special role of **endorser**. Every transaction invoking particular chaincode needs to be endorsed before being committed. Each chaincode might specify an endorsement policy that defines the necessary and sufficient conditions for valid transaction endorsement. Such endorsement might involve multiple endorsers. In the case of deploying new chaincode through transactions, the endorsement policy is applied to the *system chaincode*. System chaincode is a system-level chaincode for management functions.
- **Orderer:** An orderer node validates the transactions based on the endorsement policy and orders the transactions into a sequence before broadcasting them into the network. Orderers provide shared communication channels to clients and peers. Clients connected to a channel may broadcast transactions on the channel which are then delivered to all peers within the channel by the orderers.

  The ordering service provided by the orderers supports multiple channels, similar to the topics of a publish/subscribe messaging pattern. Clients first connect to a channel and can then send and receive transactions. Clients connecting to one channel may be unaware of the existence of other channels. Clients can connect to multiple channels.

2.3.4 Transactions

The life cycle of a Hyperledger Fabric transaction, as shown in Fig. 2.10, is different from the life cycles discussed in Fig. 1.3 and Section 2.2.2. At a high level, the life cycle of a transaction starts from a transaction proposal being created by an

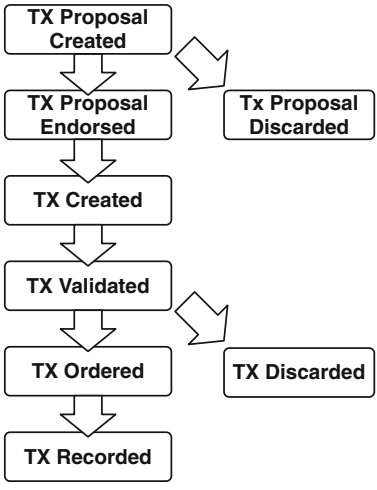


Fig. 2.10 Hyperledger transaction life cycle

application *client* and sent to specific peers as endorsers. The *endorsers* verify the signature of the initiator and execute the referred chaincode functions to prepare the transaction. The result of the execution is a set of key-value pairs read from the chaincode and a set of key-value pairs written into the chaincode. The proposal response with the signature(s) of the endorsement is sent back to the *client*.

The *client* composes the endorsements into payload of the transaction before broadcasting the transaction to an *orderer*. The *orderer* is responsible for transaction validation, ordering transactions into blocks and delivering the blocks to all *peers* on the channel. Once the *peers* receive transactions, they check the endorsement policy to ensure that the correct *peer(s)* as endorser(s) have signed the result and authenticate the signature(s) against the endorsements included in the transaction payload. The *peer(s)* ensure data integrity of the transaction through checking that the data that was read during chaincode execution has not been changed since the time of endorsement, so that the valid execution result can be committed. If the data that was read had been changed by other transactions, the transaction in the new block is marked as invalid. In this case, the *client* is alerted and needs to handle the error somehow.

2.3.5 Consensus

Based on the life cycle of transactions, consensus in Hyperledger Fabric requires the full verification of transactions and is achieved if the order and/or a set of transactions and execution results of the corresponding chaincode within a block meet the policy criteria checks. These checks take place during the life cycle of a transaction, which can be broken out into three stages: endorsement, ordering, and validation.

- Endorsement is driven by the policy defining which peer(s) endorse a certain transaction.
- Ordering accepts the endorsed transactions and orders the transactions into a sequence to be committed to the corresponding ledger.
- The blocks of transactions are ‘delivered’ to all peers on the channel. Validation checks the correctness of a set of ordered transactions within a block, considering the endorsement policy and versioning checks for data integrity.

The modular architecture design of Hyperledger Fabric allows pluggable consensus for all the three phases so that applications may use different models for endorsement, ordering, and validation according to their requirements. Other than endorsement, ordering, and validation, identity verification occurs during the consensus process.

2.4 Other Representative Blockchain Platforms

Similar to Hyperledger Fabric, Corda[10] proposed by R3[11] also has ledgers shared only between defined groups of parties. This is aimed to improve privacy and scalability by reducing the replication of data across the network. Because these systems do not implement a single global ledger, they are arguably not blockchains but nonetheless still implement a kind of distributed ledger.

Ripple[12] is a real-time gross settlement system, currency exchange, and remittance network across financial institutions. Ripple uses a common ledger that is managed by a network of independently validating servers that constantly compare transaction records. These validating servers can belong to individuals or banks.

Various techniques have been proposed to preserve privacy on blockchain. For example, Zcash[13] encrypts payment information in transactions but uses a cryptographic method to allow any node to nonetheless verify the validity of the encrypted transactions. A zero-knowledge proof construction is used to allow the blockchain network to maintain a secure ledger and enable private payment without disclosing the parties or amounts involved. Monero[14] uses other cryptographic tools to shield sending and receiving addresses and transacted amounts.

2.5 Further Reading

A more comprehensive background of blockchain functionality, features, and potential applications is discussed in Swan (2015). An early description of smart contracts on blockchain can be found in Omohundro (2014). A number of other books focus on the internal details of various blockchain platforms.

The Nakamoto proof-of-work consensus protocol, first used by Bitcoin, was introduced in the original Bitcoin paper (Nakamoto 2008).

The Ethereum yellow paper defines a detailed cost model (Wood 2015–2018) to compensate the data storage and computation power contributed by miners. The GHOST (Greedy Heaviest Observed Subtree) protocol used by Ethereum was first introduced in Sompolinsky and Zohar (2013) to tackle problems including network propagation time of blocks, short inter-block times, and miner centralization.

The Hyperledger Fabric platform is described in more detail by Androulaki et al. (2018) and in online documentation.[15]

[10] https://www.corda.net/.

[11] https://www.r3.com/.

[12] https://ripple.com/.

[13] https://z.cash/.

[14] https://getmonero.org/.

[15] https://hyperledger-fabric.readthedocs.io.

Chapter 3
Varieties of Blockchains

Since the advent of Bitcoin in 2008, a diverse range of blockchains has emerged. Blockchain has a complex internal structure and has many configurations and variants. When building applications based on blockchains, we need to systematically consider the features and configurations of blockchains and assess their impact on quality attributes for the overall systems. Since blockchains are still at an early stage, there is little product data or reliable technology evaluation available to compare different blockchains. The lack of product data and reliable technology evaluation resources makes the comparison difficult.

In this chapter, we address the manifold varieties of blockchains by presenting a design taxonomy that defines dimensions and categories for classifying blockchains and ways of using them in systems. Taxonomies have been used in software architecture to understand existing technologies. The compact framework provided by a taxonomy allows architects to explore the conceptual design space and to compare and evaluate design options. Our taxonomy captures major architecturally relevant characteristics of various blockchains and indicates their support for various quality attributes. This includes performance and quality attributes of blockchain-based systems, as well as core concerns of blockchains like decentralization and the data structure used. The taxonomy is informed by existing industrial products, technical forums, academic literature, and our own experience of using blockchains and developing prototypes.

3.1 Fundamental Properties of Blockchain

If data is contained in a committed transaction, it will eventually become in practice *immutable*. The immutable chain of cryptographically signed historical transactions provides *non-repudiation* of the stored data. Cryptographic tools also support data *integrity*, the public access provides data *transparency*, and *equal rights* allows

X. Xu et al., *Architecture for Blockchain Applications*,
https://doi.org/10.1007/978-3-030-03035-3_3

every participant the same ability to access and manipulate the blockchain. These rights can be weighted by the compute power or stake owned by the miner. A distributed consensus mechanism governs addition of new items; it consists of the rules for validating and broadcasting transactions and blocks, resolving conflicts, and the incentive scheme. The consensus ensures all stored transactions are valid and that each valid transaction is added only once.

Trust in the blockchain is achieved from the interactions between nodes within the network. The participants of blockchain network rely on the blockchain network itself rather than relying on trusted third-party organizations to facilitate transactions. These five properties (immutability, non-repudiation, integrity, transparency, and equal rights) are the main properties supported in existing blockchains.

3.2 Decentralization

Decentralization is one of the distinguishing capabilities of blockchain technology, but there are various aspects and varieties of decentralization. Decentralization devolves responsibility and capability from a central location or authority. In a centralized system, all users rely on a central authority to mediate transactions. For example in a bank, customers rely on the bank's systems to correctly adjust their account balances when a bank transfer occurs. A central authority could manipulate the whole system, including by directly updating backend databases or by upgrading the software that implements the system. Thus, a central authority is a single point of failure for a centralized system. In contrast, a fully decentralized currency system like Bitcoin allows people to reach agreement on who owns what without having to trust each other or a separate third-party. Such a system is highly available since every full node in Bitcoin network downloads every block and transaction, checks them against Bitcoin's core consensus rules, and provides functionality to process transactions. There are currently more than 9000 nodes in the Bitcoin network,[1] although not all are full nodes that form the backbone of Bitcoin.

Table 3.1 represents a spectrum of (de)centralization, from full centralization to full decentralization. The column 'fundamental properties' refers to the five properties discussed in Section 3.1. In a system it is possible that some components or functions are decentralized while others are centralized.

There are two types of centralized systems. In the first there is a monopoly service provider, including governments and courts within a jurisdiction, and business monopolies. In the other type, there are competing alternative providers, such as banks, online payments, or cloud computing providers. Any centralized system is a single point of failure for its users. However, where there are alternative providers, the failure of a single service provider only affects its users. Users may switch providers or may be able to use multiple providers.

[1] https://bitnodes.21.co/nodes/.

Table 3.1 (De)centralization with an indication of their relative impact on quality properties (⊕, less favourable; ⊕⊕, neutral; ⊕⊕⊕, more favourable)

Design decision	Option	Impact			
		Fundamental properties	Cost efficiency	Performance	#Failure points
Fully centralized	Services with a single provider (e.g. governments, courts)	⊕	⊕⊕⊕	⊕⊕⊕	1
	Services with alternative providers (e.g. banking, online payments, cloud services)				
Partially centralized and partially decentralized	Permissioned blockchain with permissions for fine-grained operations on the transaction level (e.g. permission to create assets)	⊕⊕	⊕⊕	⊕⊕	*
	Permissioned blockchain with permissioned miners (write), but permission-less normal nodes (read)				
	Off-chain transaction protocols				
Fully decentralized	Permission-less blockchain	⊕⊕⊕	⊕	⊕	Majority (nodes, power, stake)

At the other end of the spectrum, fully decentralized systems include permission-less public blockchains, such as Bitcoin and Ethereum. Permission-less public blockchains are completely open: new users can at any time join the network, validate transactions, and mine blocks. Decentralized systems using anonymous validators need to protect against *Sybil attacks*, where attackers create many hostile anonymous nodes. Bitcoin partly guards against this through its proof-of-work mechanism, so that it is not the total number of nodes that is important for integrity but rather the total amount of computational power. While it is easy for an attacker to create anonymous nodes, it is not easy for them to amass large amounts of computational power. Any system can be defeated if an attacker controls a majority of authority (nodes, computational power, or stakeholding). Game-theoretic attacks can change this threshold, requiring a higher (e.g. 66%) majority to maintain integrity. There is a spectrum of possibilities between centralization and decentralization. There are two dimensions to classify a blockchain, including *permission* and the type of *deployment*. These two dimensions are discussed in the next two subsections.

Another hybrid approach is the use of off-chain transaction protocols to progress transactions between parties and then later to reconcile the effects of those protocol executions on-chain. The Bitcoin *Lightning network*[2] moves some transactions off-chain by establishing a multi-signature transaction between two participants as a micropayment channel to transfer value off-chain. Once both sides wish to close the micropayment channel and finalize the value transfer, a transaction is submitted to the global Bitcoin blockchain. Such bidirectional channels can be connected to establish a payment network leveraging Bitcoin. The intermediate transactions occurring in the payment channel are not included in the blockchain. Raiden[3] is a similar project on Ethereum, using its smart contract facilities.

3.2.1 Permission

Instead of anonymous public participation, a blockchain may be permissioned in requiring that one or more authorities act as a gate for participation. This may include permission to join the network (and thus read information from the blockchain), permission to initiate transactions, or permission to mine. Some permissioned blockchains, e.g. MultiChain,[4] allow more fine-grained permissions, such as the permission to create assets. Permissioned blockchain networks include Ripple[5] and Eris.[6] The code for public blockchains can also be deployed on private

[2] https://lightning.network/.

[3] https://github.com/raiden-network/raiden.

[4] http://www.multichain.com/.

[5] https://ripple.com/.

[6] https://monax.io.

networks to create a kind of permissioned blockchain using network access controls. Permission information can be stored either on-chain or off-chain.

Permissioned blockchains may be especially suitable in regulated industries. For example, banks are required to establish the real-world identity of transacting parties to satisfy Know Your Customer (KYC) regulation. In contrast, a transaction on a permission-less blockchain across jurisdictional boundaries can circumvent this and undermine regulatory controls. Permissioned blockchains may be able to better control access to off-chain information about real-world assets.

There are often trade-offs between permissioned and permission-less blockchains including transaction processing rate, cost, censorship resistance, reversibility, finality, and the flexibility in changing and optimizing the network rules. The suitability of a permissioned blockchain may also depend on the size of the network. Nonetheless, the permission management mechanism may itself become a potential single point of failure, not just operationally but also from a business perspective.

3.2.2 Deployment

When using a blockchain, there are different types of deployments, including public blockchain, consortium/community blockchain, or private blockchain. An overview is given in Table 3.2.

Most digital currencies use public blockchains, which can be accessed by anyone on the Internet. Using a public blockchain results in better information transparency and auditability but sacrifices performance and has a different cost model. In a public blockchain, data privacy relies on encryption or cryptographic hashes.

A consortium blockchain is typically used across multiple organizations. The consensus process in a consortium blockchain is controlled by pre-authorized nodes. The right to read the blockchain may be public or may be restricted to specific participants. In a private blockchain network, write permissions are often kept within one organization, although this may include multiple divisions of a single organization.[7]

Whether using a consortium blockchain, private blockchain, or permissioned public blockchain,[8] a permission management component will be required to authorize participants within the network. Private blockchains are the most flexible for configuration because the network is governed and hosted by a single organization. Many blockchain platforms support deployment as consortium blockchains or private blockchains, e.g. MultiChain and Eris.

[7] There is a grey area between consortium blockchains and private blockchains, and the differences may be more administrative than technical. Nonetheless we distinguish them here because at their extremes they have architectural differences.

[8] Ripple can arguably be seen as a permissioned public blockchain.

Table 3.2 Blockchain deployment (⊕, less favourable; ⊕⊕, neutral; ⊕⊕⊕, more favourable)

	Impact			
Deployment option	Fundamental properties	Cost efficiency	Performance	Flexibility
Public blockchain	⊕⊕⊕	⊕	⊕	⊕
Consortium/community blockchain	⊕⊕	⊕⊕	⊕⊕	⊕⊕
Private blockchain	⊕	⊕⊕⊕	⊕⊕⊕	⊕⊕⊕

Table 3.3 Ledger structure (⊕, less favourable; ⊕⊕, neutral; ⊕⊕⊕, more favourable)

	Impact			
Option	Fundamental properties	Cost efficiency	Performance	Flexibility
Global list of blocks (Bitcoin)	⊕⊕⊕	⊕	⊕	⊕
Global DAG of blocks (Hashgraph)	⊕⊕	⊕⊕	⊕⊕	⊕⊕
Global DAG of transactions (IOTA)	⊕⊕	⊕⊕	⊕⊕	⊕⊕
Restricted shared ledgers (Corda)	⊕	⊕⊕⊕	⊕⊕⊕	⊕⊕⊕

3.3 Ledger Structure

The ledger can be structured in different ways; Table 3.3 provides an overview. In Bitcoin, the history of all transactions is captured in the blockchain structure. This is a single global list (chain) of lists (blocks) of transactions, as discussed in Chapters 1 and 2. Bitcoin nodes actually record the blockchain as a tree of blocks, where shorter branches attached to the main chain represent alternative competing histories. However, the tree data structure is relevant mainly for the nodes operating the blockchain and determining consensus; under the logical view from a user's perspective, the blockchain is a list of blocks. This is similar for Ethereum.

Other blockchain and distributed ledger systems have different data structures. For example, the logical view of transactions recorded in Hashgraph[9] is based on a directed acyclic graph (DAG) of blocks, rather than a list. Somewhat similarly, IOTA[10] also uses a DAG but of individual transactions rather than blocks of transactions.

These systems all maintain a single global transaction history. Other distributed ledger systems such as Hyperledger Fabric and Corda have been proposed where there are essentially many small ledgers, shared only between parties of interest

[9] https://www.hederahashgraph.com/.
[10] https://www.iota.org/.

who are authorized to view the transactions recorded in those ledgers. For the Corda distributed ledger, the abstract logical view of transaction history is of a global graph of transactions. However, transactions are only distributed to parties of interest; special agents (notaries) can be used to further limit the distribution of transactions while attesting to the integrity of unseen parts of the transaction graph. So although there is notionally a global graph of transactions, the view that most parties see is a collection of small ledgers, each shared with their related business contacts. Hyperledger Fabric is somewhat similar, because parties also see a collection of small ledgers shared with related business contacts (via 'channels'). However, Fabric has a more rigid transaction distribution policy, isolating transactions within the channels.

3.4 Consensus Protocol

The choice of *consensus protocol* impacts security and scalability. An overview is given in Table 3.4. Once a new block is generated by a miner, the miner propagates the block to its connected peers in the blockchain network. However, miners may encounter different competing new blocks and resolve this using the blockchain's consensus mechanisms. Usually the approach is fixed for a particular blockchain; but Hyperledger Fabric deviates from this norm, as a framework with a modular architecture that caters for pluggable implementations of various consensus protocols.

The typical overall approach is called Nakamoto consensus, as introduced in Section 2.1.5. This relies on participants selecting as authoritative the longest chain of blocks they have observed at every point in time. In Bitcoin, new blocks are generated through a *proof-of-work* mechanism. Proof-of-work uses a cryptographic puzzle which is easy to verify, but solving it is difficult and takes effectively random

Table 3.4 Consensus protocol (⊕, less favourable; ⊕⊕, neutral; ⊕⊕⊕, more favourable)

Option		Impact			
		Fundamental properties	Cost efficiency	Performance	Flexibility
Security-wise	Proof-of-work	⊕⊕⊕	⊕	⊕	⊕
	Proof-of-retrievability	⊕⊕⊕	⊕	⊕	⊕
	Proof-of-stake	⊕⊕	⊕⊕	⊕⊕	⊕⊕⊕
	Practical Byzantine Fault Tolerance (PBFT)	⊕	⊕⊕⊕	⊕⊕⊕	⊕
Scalability-wise	Bitcoin-NG	⊕⊕⊕	⊕	⊕	⊕
	RBBC	⊕⊕	⊕⊕⊕	⊕⊕⊕	⊕

time. Bitcoin miners compete to solve such a puzzle for each block, using large amounts of computer power (and hence electricity) to increase their chances of winning the competition for the block. The investment required by miners for this acts to align their incentives with the good operation of the overall system. There are various proof-of-work mechanisms, such as Ethash[11] used by Ethereum and Hashcash[12] used by Bitcoin. The work done in proof-of-work systems can sometimes be put to good use. For example, the mechanism in Primecoin[13] generates prime number chains which are of interest to mathematical research. Permacoin uses 'proof-of-retrievability' to repurpose Bitcoin's mining resources to distributed storage of archival data.

Proof-of-stake is an alternative mechanism for Nakamoto consensus, which selects the next mining node based on the control of the native digital currency of the blockchain network. For example, the miners in Peercoin[14] need to prove the ownership of a certain amount of Peercoin currency to mine blocks. Thus, proof-of-stake naturally aligns the incentives of digital currency holders in the blockchain with the good operation of the blockchain. There are various proof-of-stake protocols, e.g. Tendermint[15] used in Eris and Casper[16] for Ethereum. These have different design goals, favouring some non-functional properties over others. Proof-of-stake does not necessarily select the next miner based on largest stakeholding, e.g. Nxt[17] also uses a random factor, and Peercoin combines randomization and coin age. BitShares[18] uses *delegated proof-of-stake*, where the accounts may delegate their stake to other accounts, rather than participating in the process of validating transactions directly. The representatives take turns in a round-robin manner, signing blocks. Compared with proof-of-work, proof-of-stake is more cost-efficient because much less computational power is used in mining and latency is also shorter. However, passive holding of assets may become harder.

The *Practical Byzantine Fault Tolerance (PBFT)* protocol has been applied for consensus in permissioned blockchains, e.g. in Stellar.[19] PBFT ensures consensus despite arbitrary behaviour from some fraction of participants. Compared to Nakamoto consensus, it is a more conventional approach within distributed systems. Roughly speaking, PBFT-based blockchains offer a much stronger consistency guarantee and lower latency but for a smaller number of participants. The core of Tendermint is also a PBFT protocol but uses a proof-of-stake mechanism to prevent Sybil attacks. PBFT requires that all participants must agree on the list of

[11] https://github.com/ethereum/wiki/wiki/Ethash.

[12] https://en.bitcoin.it/wiki/Hashcash.

[13] http://primecoin.io/.

[14] http://peercoin.net/.

[15] http://tendermint.com/.

[16] https://github.com/ethereum/casper/.

[17] https://nxt.org/.

[18] https://bitshares.org/.

[19] https://www.stellar.org/.

participants in the network. Thus, the protocol is normally only used in permissioned blockchains.

Some new protocols have been proposed to improve *scalability*. Bitcoin-NG decouples Bitcoin's operation into two planes: leader election and transaction serialization. Once a leader is selected, it is entitled to serialize transactions until the next leader is selected. Thus, the leader election in Bitcoin-NG is forward-looking and ensures that the system is able to continually process transactions. Another new protocol is used in the Red Belly Blockchain (RBBC). This algorithm is a kind of democratic Byzantine consensus approach in not requiring leader nodes. The approach starts with submitted transactions being collected by a set of proposers. These nodes collectively decide on a proposed set of transaction to send to a verifier nodes, who enforce consensus using hashes exchanged for the proposed sets of transactions.

3.5 Block Configuration

Block configuration concerns options for the size (number/complexity of transactions) allowed in blocks and the frequency by which blocks are generated. These choices can impact *scalability* in terms of transaction processing rate. An overview is given in Table 3.5.

One configuration change would be to adjust mining difficulty to shorten the time required to generate a block, thus reducing latency and increasing throughput. However, a shorter inter-block time would lead to an increased frequency of forks. Ethereum has a much shorter inter-block time (10–20 s) than Bitcoin, while still using Nakamoto consensus and proof-of-work. The increased frequency of forks ('uncle blocks' in Ethereum's terminology) leads to users waiting for more confirmation blocks than in Bitcoin, though still achieving overall lower transaction latency.

Another important block configuration parameter concerns block size. Depending on the blockchain used, this is specified differently, e.g. as block size limit in Bitcoin (data size in MB) or as block gas limit in Ethereum (limiting the complexity of the contained transactions). For example, there are some proposals for Bitcoin to increase its block size from 1 to 8 MB, to include more transactions into a block and

Table 3.5 Block configuration (⊕, less favourable; ⊕⊕, more favourable)

	Impact			
Option	Fundamental properties	Cost efficiency	Performance	Flexibility
Original block size and frequency	⊕⊕	n/a	⊕	n/a
Increase block size/decrease mining time	⊕	n/a	⊕⊕	n/a

thus increase maximum throughput. The decision on the size of blocks is subject to a trade-off between speed of replication, inter-block time, and throughput and works as follows. When a new block has been proposed, processing nodes need to select a set of transactions from the transaction pool/mempool and validate and execute those. This cannot be done before observing the latest block, because the state changed as a result of the new block and may render some transactions invalid or alter their effects. Once that is complete, the block can be formed, and, in the case of proof-of-work consensus, mining can start. On the one hand, if the block can be too big or too complex, transaction processing may take too much time. Take the extreme example of having no limit; then, the system could be subject to a DoS attack by flooding it with transactions, such that the inter-block time would rise to unacceptable levels. Very big blocks also take longer to replicate among the full nodes. On the other hand, high limits can result in higher throughput. For these reasons, block limits should be set with care in private and permissioned networks. On the public Bitcoin blockchain, the long-time limit of 1 MB sparked significant controversy[20] and led to an effective increase to 2–4 MB. Public Ethereum's block gas limit has changed a number of times (see also Section 11.6.2) and is about eight million gas at the time of writing.[21] On public proof-of-work blockchains, high block limits also increase the risk of empty blocks. Consider the case where miner *A* tries to include many transactions and miner *B* tries to mine empty blocks. While *A* is processing transactions, *B* is already working on its proof-of-work, thereby increasing its relative chances to find a new block first. If block limits and block mining rewards are high, it might actually be economical to mine as many empty blocks as possible. Unfortunately, that also deteriorates the value of the network, because now it does not process new transactions anymore.

3.6 Auxiliary Blockchains

When building and deploying a new blockchain, it might be combined with or built on an existing blockchain, thus forming an *auxiliary blockchain*. Different strategies can be used to achieve security and scalability. An overview is given in Table 3.6.

For *security*, the new blockchain can be aligned with public blockchains, utilizing existing infrastructure, resources, and trust. The first option is *merged mining*, which reuses the mining power of an existing public blockchain to mine and secure the new blockchain. In this case, a proof-of-work found by a miner of the public blockchain is used by both blockchains. First, the miner produces a transaction set for both blockchains. The hash of the block produced for the new blockchain is added to the public blockchain. Then, once the miner finds a proof-of-work solution at the difficulty level of either blockchain, the proof-of-work is combined with the

[20] https://en.bitcoin.it/wiki/Block_size_limit_controversy.

[21] https://etherscan.io/chart/gaslimit.

Table 3.6 Auxiliary blockchains (⊕, less favourable; ⊕⊕, neutral; ⊕⊕⊕, more favourable)

Option		Impact			
		Fundamental properties	Cost efficiency	Performance	Flexibility
Security-wise	Merged mining	⊕⊕⊕	⊕⊕	⊕	⊕
	Hook into popular blockchain at transaction level	⊕⊕	⊕	⊕⊕	⊕⊕⊕
	Proof-of-burn	⊕	⊕	⊕⊕⊕	⊕⊕
Scalability-wise	Sidechains	⊕⊕⊕	⊕	⊕	⊕
	Multiple private blockchains	⊕	⊕⊕⊕	⊕⊕⊕	⊕⊕⊕
	Mini-blockchain	⊕⊕	⊕⊕	⊕	⊕⊕

transaction set and submitted to the corresponding blockchain. Namecoin is the first blockchain that uses merged mining with the Bitcoin blockchain. Merged mining reuses an established blockchain network. It might be difficult initially to persuade the miners of an existing blockchain to join a new blockchain network.

A more loosely coupled way is to *hook* the new blockchain into a public blockchain, by periodically adding hashes of the new blockchain to transactions of the public blockchain. For instance, Factom[22] anchors into the Bitcoin blockchain by submitting a transaction to the Bitcoin blockchain every 10 min, with the current hash of the Factom blockchain.

The third option is *proof-of-burn*. The purpose of proof-of-burn is to verifiably destroy tokens on the existing chain rather than minting new tokens on the new chain. To 'transfer' tokens from a public blockchain to the new blockchain, the participants need to provide proof that their tokens were sent to a verifiably unspendable address. The burnt tokens, originally mined by proof-of-work, represent the corresponding computational power. Proof-of-burn can be used for bootstrapping a new cryptocurrency, e.g. Counterparty,[23] as it ensures serious commitment.

Auxiliary blockchains can also be used to improve *scalability*. Rather than using a unique chain to record all types of transactions, multiple blockchains can be used to isolate information of separate concerns and with different characteristics and therefore improve scalability. Different mechanisms have been proposed to support interaction across multiple blockchains. One of the mechanisms is to use an off-chain hash lock. In the Bitcoin ecosystem, using a hash lock with *contracts* can enable *atomic cross-chain trading*,[24] which allows one cryptocurrency (e.g. the Bitcoin cryptocurrency, BTC) to be traded for another cryptocurrency (e.g.

[22] http://factom.org/.

[23] http://counterparty.io/.

[24] https://en.bitcoin.it/wiki/Atomic_cross-chain_trading.

tokens on a Bitcoin sidechain). This mechanism is also applicable in the Ethereum ecosystem.[25]

The first option for scalability is to use sidechains. Sidechaining is a mechanism that allows tokens of one blockchain to be securely transferred and used in another blockchain; eventually, they can be moved back to the original chain securely. The original chain is called *main chain*, and the one that accepts the tokens from the original chain is called *sidechain*. The second option is to have multiple private chains, where each of the private chains could link with a public blockchain. With sidechains, there is a layer of separation between two blockchains, which means that the main chain can be protected from issues or damages on the sidechains. Sidechains can help to build a blockchain ecosystem based on a popular main blockchain, without significantly increasing the load on the main chain. However, the clients of sidechains may become complex, because they typically need to be able to process transactions from the main chain and the sidechain.

There are two ways of sidechaining: unilaterally pegged sidechain and bilaterally pegged sidechain. For a unilateral (or one-way) peg, the interaction is only from the main chain to the sidechain, e.g. through proof-of-burn. For a bilateral peg, the interaction is bidirectional. One mechanism to secure bilateral pegged sidechains is essentially a voting system, where a group of custodians cast votes on when to lock and unlock tokens on one blockchain and where to send tokens on the other blockchain. The first option is to have an exchange holding the locked tokens from one blockchain and the unlocked equivalent tokens from the other blockchain. The exchange would locally enforce the promise of locking the tokens from one blockchain before unlocking the tokens of the other blockchain. This design introduces a central trusted third-party to control the exchange. A better option is to have a group of notaries control a multi-signature wallet, where a majority has to approve unlocking tokens. This is more decentralized than the first option but still centralizes control to a degree. To achieve better decentralization, the notaries could be from different jurisdictions and geographies with good reputation and good security.

The full nodes of most blockchain networks need to keep all historical transactions and the state of blockchain network, which requires sizeable storage space. For example, Bitcoin and Ethereum require more than 200 GB[26] and 600 GB[27] of storage space, respectively, at the time of writing, and these sizes keep growing. To reduce the storage burden of blockchain participants and address other scalability concerns, applying the concept of *sharding* to blockchain has been proposed. Sharding means to divide the state of blockchain into pieces. The participating blockchain nodes only hold data of some shards instead of the complete blockchain data structure. There are two types of sharding, including transaction sharding and

[25] https://dappsforbeginners.wordpress.com/tutorials/two-party-contracts/.

[26] https://bitnodes.earn.com/dashboard/bitcoind/.

[27] https://bitinfocharts.com/ethereum/.

state sharding. Elastico and Zilliqa[28] support transaction sharding. Ethereum 2.0[29] plans to improve scalability of its public blockchain through sharding based on structuring the network into two layers.

Instead of keeping all transaction information, a *mini-blockchain* scheme proposed by Cryptonite[30] periodically forgets old transaction history. The Cryptonite network maintains an account tree that holds the balance of all addresses and a separate proof chain that stores all the historical block headers. The account tree is updated according to the transactions, and after a period of time, the transactions are forgotten by the network. Neither off-chain transactions nor the mini-blockchain stores all the transactions on the blockchain. Thus, both sacrifice the fundamental properties of blockchain. The mini-blockchain saves space by forgetting historical transactions, but its performance is not necessarily better because the consensus mechanism is still the same.

3.7 Anonymity

Although the Bitcoin blockchain is perceived to be anonymous, research has shown that Bitcoin transactions can be linked to compromise the anonymity of Bitcoin users. Different techniques have been proposed to preserve anonymity on blockchain. Zcash,[31] also called Zerocash or Zerocoin, encrypts the payment information in the transactions and uses a cryptographic method to verify the validity of the encrypted transactions. A zero-knowledge proof construction is used to allow the blockchain network to maintain a secure ledger and enable private payment without disclosing the parties or amounts involved.

Mixing services offer an alternative method for anonymization. A mixing service groups several transactions together so that a payment contains multiple input addresses and multiple output addresses. Anonymity is preserved because it is hard to track which output address is paid by which input address. To further improve the way that mixing service operates, a series of mixing services can be linked sequentially. If the mixed transactions are uniform in value, the traceability between input and output addresses is minimized. Uniform values can be achieved by using standardized denominations, similar to bank notes and coins in traditional cash. A centralized mixing service requires a third-party to operate, e.g. CoinJoin[32] and Blindcoin. Distributed mixing services, on the other hand, do not rely on a single third-party, e.g. CoinSwap.[33] Some blockchains have a kind of native, built-in

[28] https://zilliqa.com/.

[29] https://github.com/ethereum/wiki/wiki/Sharding-FAQs.

[30] http://cryptonite.info/.

[31] https://z.cash/.

[32] https://bitcointalk.org/index.php?topic=279249.0.

[33] https://bitcointalk.org/index.php?topic=321228.0.

mixing service, including Dash and Monero. Dash pre-anonymizes funds of users through mixing rounds, so that the funds can later be spent without delay.[34] In contrast, Monero uses ring signatures, such that the sender of a transaction cannot be identified among a group of possible senders.

3.8 Incentives

Blockchains and their applications (especially on public blockchains) introduce financial incentives in the cryptocurrencies of the respective networks. Incentives are paid to make miners to join the network, validate transactions, generate blocks, and (where applicable) execute smart contract functions correctly. For example, in Bitcoin, miners have two incentives: the reward for generating new blocks and the fees associated with transactions. Miners in Ethereum also charge a fee to execute smart contracts. Enigma[35] has a fixed price for storage, data retrieval, and computation within the network. Enigma also requires a security deposit for nodes to join the network. If a node is found to lie, its deposit will be split among the honest nodes.

3.9 Summary

Blockchain platforms can have various configurations and design options. Using blockchain in different scenarios requires the comparison of blockchain options and products with different implementations and configurations. In this chapter, we discussed a taxonomy of blockchain systems. The taxonomy can be used when comparing blockchains and assist in the design and evaluation of software architectures using blockchain technology. Our taxonomy captures major architectural characteristics of blockchains and the impact of different decision decisions. This taxonomy is intended to help with important architectural considerations about the performance and quality attributes (e.g. availability, security, and performance) of blockchain-based systems.

3.10 Further Reading

This chapter is partly based on our earlier works (Xu et al. 2017).

Taxonomies have long been used in the software architecture community to understand existing technologies (see, e.g. Mehta et al. 2000; Gorton et al. 2015).

[34] https://docs.dash.org/en/latest/introduction/features.html#privatesend.

[35] https://www.media.mit.edu/projects/enigma/overview/.

From a software architecture perspective, blockchain can also be characterized as a software connector (Xu et al. 2016), which has a complex internal structure and many configurations and variants. Blockchain is a decentralized system that can be defeated unless there is a majority of honest or favourable authority (computational power, stakeholding, etc., depending on the consensus mechanism). Eyal and Sirer (2018) show that game-theoretic attacks can change this threshold for proof-of-work, requiring a higher (e.g. 66%) majority to maintain integrity and prevent double-spending attacks. More definitions of different types of blockchain and discussion on the trade-offs between them can be found in Swanson (2015) and Buterin (2015).

Nakamoto consensus provides probabilistic immutability. There is always a chance that the most recent few blocks get replaced by a competing chain fork. The impact of inter-block time on the frequency of forks is discussed in Decker and Wattenhofer (2013). A detailed comparison between proof-of-work and proof-of-stake can be found in Gervais et al. (2016). Permacoin's 'proof-of-retrievability' is discussed in Miller et al. (2014). Discussion on PBFT-based blockchains can be found in Vukolić (2015). The Red Belly Blockchain (Crain et al. 2017) uses a new kind of democratic Byzantine consensus protocol. Some protocols have been proposed to improve scalability, for example, Bitcoin-NG (Eyal et al. 2016) and the Bitcoin Lightning network (Poon and Dryja 2016).

More information on sidechaining can be found in Back et al. (2014). Blockchains that apply sharding technology are discussed in Luu et al. (2016) and Danezis and Meiklegohn (2016).

Detail of Blindcoin can be found in Valenta and Rowan (2015).

Chapter 4
Example Use Cases

To convey a more concrete picture of applications of blockchain, this chapter presents four exemplar use cases which illustrate some of the techniques and considerations discussed in the previous chapters. These use cases are also used as running examples throughout the book, but their details are not strictly necessary for understanding later parts of the book. For every use case, we give a brief background and describe their key non-functional requirements.

4.1 Agricultural Supply Chains

In manufacturing, retail, and agricultural industries, supply chains are critical in the movement of goods and services across organizational boundaries. Supply chain contracts are complex, dynamic, multiparty arrangements, with regulatory and logistical constraints. They often cross jurisdictional boundaries. The information exchange in a supply chain can be as important as the physical exchange of goods. For example, customs inspections would not start until both the physical goods and the information about those goods are present. Confidence in supply chain documentation can expedite customs and biosecurity processes, reduce risk and insurance costs, and be used as leverage in trade finance. Payments are made between parties at many points in the supply chain.

For agricultural food products, being able to tell where ingredients were grown and how products were processed and distributed can be important in establishing confidence in food safety, creating and building high-quality brands, reducing fraud, and improving supply chain efficiency. There are many stakeholders in an agricultural supply chain, ranging from producers to transport providers, sorting/processing facilities, wholesalers, distributors, retailers, and consumers. In international supply chains, there are also stakeholders related to customs and biosecurity. A simplified configuration of some stakeholders and functions is shown in Fig. 4.1.

X. Xu et al., *Architecture for Blockchain Applications*,
https://doi.org/10.1007/978-3-030-03035-3_4

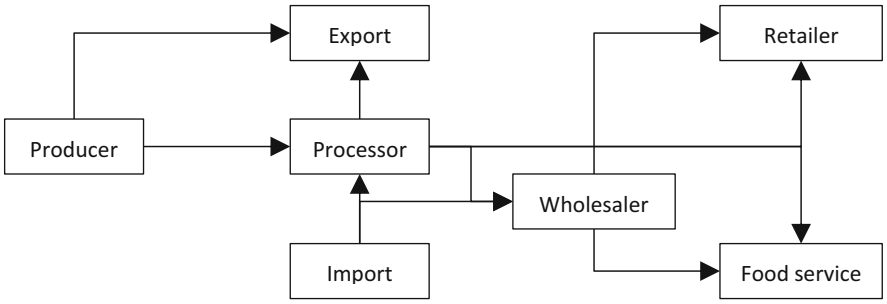


Fig. 4.1 Stakeholders in a simple agricultural supply chain. © 2017 by the Commonwealth Scientific and Industrial Research Organisation, reprinted with permission

The information systems supporting supply chains normally reside at the individual supply chain participants and are integrated to varying degrees, i.e. from no digital integration to machine-readable barcodes that can be understood by a number of participants, through to full system integration with digital message exchanges.

4.1.1 Key Non-functional Requirements

- Interoperability: A huge challenge in logistics is to coordinate information exchange across the many different kinds of goods, modes of transport, and information systems. Individual shipments can be aggregated into larger consignments, which means tracing information about the status of goods can require integration of different interlinked information sources.
- Latency: The exchange of physical goods must sometimes wait upon exchange of documentation associated with the delivery. Information exchange should not introduce significant additional delays at these points.
- Integrity: Supply chain quality and provenance require that information about goods and supply chain events cannot be falsified or created without proper authority.
- Confidentiality: Some information in supply chain documentation should be held commercial-in-confidence. Even metadata can expose aggregate trade flows which can be commercially sensitive. However, because of long supply chains and the use of subcontractors, parties' interests in information about supply chain events may extend beyond the parties directly involved in that event. Balancing transparency and commercial confidentiality is a complex business model problem.
- Scalability: There are many supply chain processes in progress at any time across a large number of different parties. Each process instance creates a large number of events, although not all events are relevant to all participants. A system must scale to handle the total throughput of transactions, with parties using resources in proportion to their level of involvement in the process.

4.1.2 Conventional Technology

Traditionally, supply chain information is recorded separately by each entity in the chain. Each participant only sees the information they are a direct party to. As supply chain systems have become more digitized, information sharing has become more common. Standards such as GS1's EPCIS (Electronic Product Code Information Services) define uniform schemes for representing supply chain events. This can help parties in a supply chain to record and exchange information.

Figure 4.2 depicts a design for a supply chain system using EPCIS and other data with conventional technologies. All EPCIS data is sent to a central event aggregation server for an agreed portion of the supply chain. A group of supply chain participants agree on a trusted party to operate and control access to the aggregation server. Note that this design would be an advance over many current supply chain systems, but it has been implemented in some industrial settings. Note also that the centralized server creates a risk as a single point of failure, either for operational reasons or for business reasons. (Business reasons may include complete business failures or perhaps merely unfavourable changes in pricing or terms of use.)

Supply chain events are not the only type of information that needs to be exchanged. Other documents may include letters of credit, bills of lading, booking confirmations, arrival notices, container releases, terminal load lists, delivery orders, tax invoices, and so on. These other types of documents are normally kept locally to the systems of the different supply chain participants and exchanged directly using

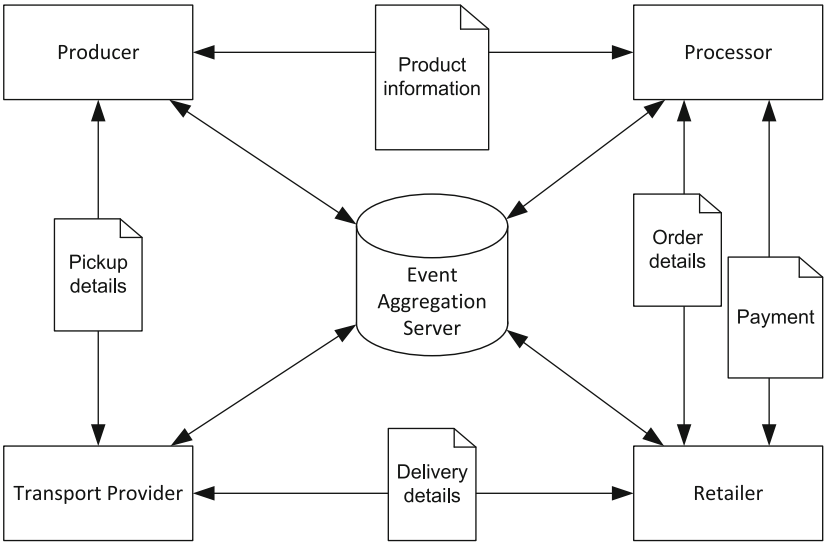


Fig. 4.2 Model of supply chains using conventional event aggregation server and point-to-point integration. © 2017 by the Commonwealth Scientific and Industrial Research Organisation, reprinted with permission

point-to-point integration between parties. Currently, it can be hard to guard against fraud that uses forged or tampered documents.

4.1.3 A Blockchain Solution

One possible alternative solution using blockchain is to control the execution of the process of a supply chain using smart contracts. A group of participants that want to implement a shared supply chain process first agree on a design for the collaborative process that regulates how their interactions should take place. The controls for this process are implemented using smart contracts, and the participants coordinate the progress of that process by calling those smart contracts in turn. The smart contract can enforce the process as follows. First, it can reject messages if they arrive at the wrong point in the process. Second, messages are only accepted from the participant who is authorized to send them. Third, conditions can be specified within the process model and can be executed in smart contract code directly. So particular process branches will be automatically activated when their conditions are met.

Consider an example: containerized export of wine from a rural Australian producer. This starts when the producer initiates a shipment and ends when the container is on a ship. One process instance deals with exactly one container, and once the container number is assigned, it can be used as an identifier of the process instance. Figure 4.3 shows the process model.

These smart contracts can be generated automatically from process models, as we discuss in Chapter 8. In the resulting system, the supply chain participants interact with each other by sending messages through the blockchain. To facilitate interaction through blockchain, so-called trigger components act as bridges between the blockchain and enterprise applications. The trigger can translate conventional service calls to blockchain transactions and vice versa. This can keep the implementation cost relatively low. For message formats, we can use the same standards as in the conventional design, i.e. GS1 EPCIS.

4.1.4 Non-functional Property Discussion

Interoperability Both designs use the GS1 EPCIS standard for events. The first design requires point-to-point integration between any two participants for other documents. Extending the supply chain to a new participant requires integration of that participant's system with all participants that need to exchange documents directly with the new participant. The second design requires the same amount of integration initially: the data formats also need to be agreed upfront. However, each new participant only needs to integrate their systems once, with the blockchain process, and thus the overall integration burden is reduced.

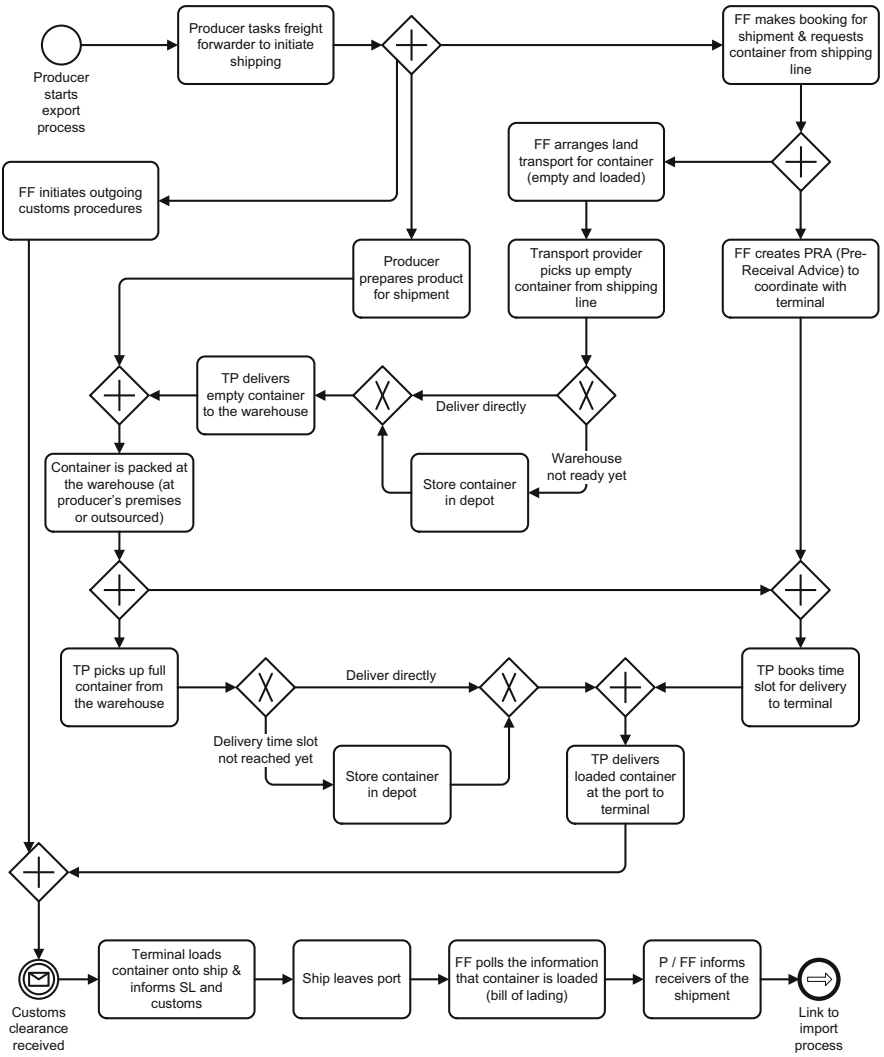


Fig. 4.3 Process model of an agricultural export supply chain process. *FF* freight forwarder, *TP* transport provider, *P* producer. Notation: BPMN. © 2017 by the Commonwealth Scientific and Industrial Research Organisation, reprinted with permission

Latency Supply chains typically involve the physical movements of goods, so many latency requirements on information transfer are usually on the order of minutes to hours. Neither of the designs should suffer from latency exceeding these time frames. However, at points of handover of goods, there may be low latency requirement for confirmation of receipt of goods. Commit times on public blockchains are likely to be too long for this, but it may be possible to instead

provide cryptographically signed receipts off-chain, with the delivery agent able to lodge those to the blockchain at a later time.

Integrity The first design relies on a trusted party to operate the aggregation server and is subject to the possibility of manipulation with a low chance of detection. Integrity is a strong inherent feature of blockchains: information captured as part of committed transactions would be exceedingly hard to change. If large blocks of data (such as photos or video) need to be stored, this could be done off-chain, with integrity preserved by storing a cryptographic hash of this data on-chain. This allows detection of alterations or corruption of the off-chain data, but increases design complexity.

Scalability In both designs, each party has to deal with the scalability of their own enterprise applications, which we do not discuss here. Instead, we focus on scalability of the components shared by all parties. In the first design, this is the central aggregation server. If all participants publish all event data for item movements, this might become a bottleneck. There are many design options available to address scaling of web-based centralized information systems, including filtering to only publish events that are relevant for other parties and using load balancing services to federate data access across multiple aggregation servers.

In the second design, the component shared by all participants is the blockchain. Scalability of reading from the blockchain can be good, since each participant can hold their own full copy of the blockchain. For writing new transactions and smart contract method calls, scalability is currently limited on public blockchains. For this design, we propose using a *consortium blockchain*, where transaction volumes can be controlled and where other technical options for block formation and consensus are available to improve performance. As with the first design, only relevant events should be stored on-chain. In the second design, communication is also limited to the messages exchanged as part of the collaborative process execution. Throughput scalability can be achieved by careful design and performance tuning. As discussed in the previous chapter, specific types of blockchains that do not use Nakamoto consensus have been designed for private or consortium blockchains with high scalability requirements.

Confidentiality Confidentiality requirements for supply chain data are not the same across industries or participants. This affects both designs: for a specific supply chain and a specific set of participants, the confidentiality requirements need to be formulated and analysed, and potentially the design needs to be adapted accordingly. The main trade-off is between the benefits of sharing data within the group of collaborators—visibility and cross-party optimizations are impossible without that—retaining confidentiality between competitors where needed. Supply chain information can be commercial-in-confidence. This may include the identities of participants, trade volume, prices, and delivery times.

While it is possible to restrict access to the aggregation server in the first design and the consortium blockchains in the second design, it should be expected that

for some supply chain roles multiple competing participants have access to the same system. Even a private blockchain does not protect commercial-in-confidence information. Unless the supply chain is entirely vertically integrated within one organization, competitors will be sharing access to information on the blockchain. The only way to prevent that is by setting up a separate aggregation server or blockchain for each group of parties. That is, switching transport providers would require setting up a separate system, which would not only be tedious and resource-intensive, but would also severely hamper the analysis of supply chain data across specific instances. Alternative distributed ledger technologies, such as R3's Corda or Hyperledger Fabric, natively support the creation of separate ledgers for related parties, e.g. through Fabric's channels. However, they still suffer from the second issue: the lack of visibility hampers global analysis and optimization.

Data stored on a blockchain is readable to all participants of that blockchain. Confidential data can be encrypted, and keys can be exchanged between supply chain participants so that only the 'right' group of participants can decrypt that data. However, this requires off-chain key exchanges and diligent handling of keys. Moreover, normally encrypted data can itself not be processed by the blockchain or its smart contracts. Thus, transfers of assets that are managed by the blockchain cannot be encrypted; and encrypted data cannot be transformed or actioned by smart contracts. New sophisticated cryptographic techniques such as homomorphic encryption and zero-knowledge proofs allow various kinds of computation or transaction validation to be performed on encrypted data, without decrypting it. These techniques are being explored for use in blockchain platforms and may provide an alternative treatment for this issue.

Finally, a confidentiality concern can arise from metadata, not just data inside the transactions. For example, the volume of interactions between parties may reveal trade volumes. It would be possible to create new account addresses for each participant and each new process instance, but the flow of assets may still be used to infer relationships between addresses, revealing aggregate trade volumes. Dummy transactions might be used to attempt to hide this. Such protection mechanisms can help, but may erode the benefit of using a blockchain. These trade-offs require careful consideration.

4.2 Open Data Registry

Registries are authoritative collections of information, usually managed centrally, often by government agencies. A registry holds information about a class of entities. Examples of such entities include individuals, businesses, species, and organizations. In Australia, familiar registries include the immunization registry, the business name registry, and land title registries. There are also well-known international registries such as the Domain Name System (DNS). Some government registries are described as 'public' and can be queried by individuals. However, query access to these registries may be limited to prevent attempts at republishing or

data mining. Unfettered data mining could threaten commercial or personal privacy and is often restricted using regulatory policies, query rate limits, and user access controls.

Some government registries contain periodically published open data. In Australia, these are published through `data.gov.au`.[1] In this use case, we specifically consider the use of blockchains for managing an open data registry of datasets, data sources, and data analytics services. This means we do not consider confidentiality or privacy issues for this use case. Blockchains provide transparency about their entire transaction history to all processing nodes. In a public blockchain, this means that the information is openly published. It is possible to run a private blockchain hidden behind a web service or other interfaces. This could limit access to the registry in a way that satisfies an appropriate access policy. However, many of the benefits of using a blockchain would be foregone in such an architecture.

For open data, the major stakeholders are data providers, data consumers, and the data registry. Data providers may include government agencies, research institutes, universities, and companies. Data providers record metadata about their datasets on the data registry and make their data available on their websites. Data consumers query to discover datasets in the data registry based on the metadata. They can then download the datasets from the data providers for analysis.

4.2.1 Key Non-functional Requirements

- Integrity: each data provider should only be able to create and change registry entries for their own datasets.
- Availability: there should be high likelihood of being able to access the registry when desired, for both data providers and data consumers. This particularly applies to national public registries, which form the basis for many other services that utilize the data from the registries.
- Read latency: data consumers may need to repeatedly query the registry while browsing and searching for relevant datasets. This may be done programmatically from a graphical user interface and so should have low latency.
- Interoperability: a registry may reference other registries to reduce duplication and errors.
- Ease of integrating new data providers: to grow the network effects of the registry as a data portal, it is important to have low barriers (time, cost, and administrative burden) to add new data providers to the registry.

[1] https://data.gov.au.

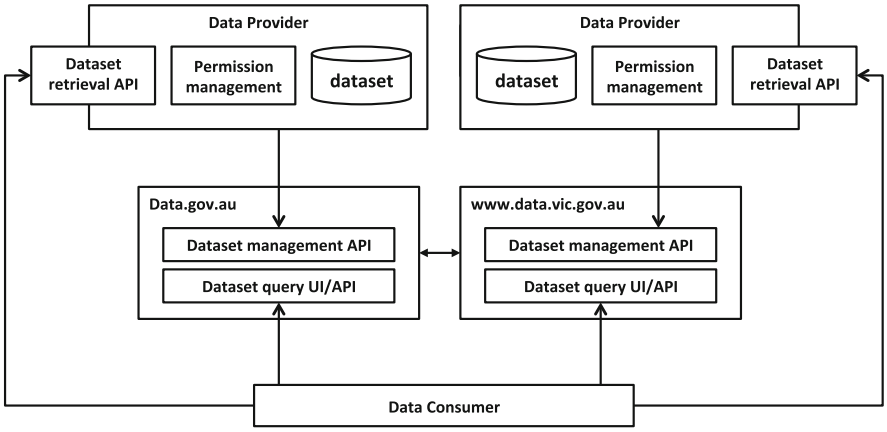


Fig. 4.4 Design for a registry using conventional technologies, operated by a single agency. © 2017 by the Commonwealth Scientific and Industrial Research Organisation, reprinted with permission

4.2.2 Conventional Technology

Data portals such as `data.gov.au` implement a dataset registry using conventional technologies such as CKAN.[2] For each portal, the CKAN software is run and managed by a single government agency. Data consumers interact with a registry to discover datasets but retrieve datasets directly from data providers. The data providers may perform some permission management for data access independently. An illustrative high-level design is shown in Fig. 4.4.

In the CKAN ecosystem, datasets in different CKAN repositories refer to each other by importing metadata from each other.

4.2.3 A Blockchain Solution

We consider a design which replaces the registry with a public blockchain. In this design there is no single agency that operates the registry. Instead the data providers independently record metadata on the public blockchain and perform their own permission management and access control for their datasets independently. Note that there may still be an agency leading governance for the registry. In this design, data consumers are required to interact directly with the blockchain, rather than with a consumer-facing user interface or API. Those consumer interfaces may be provided by commercial or personal systems. An illustrative high-level design is shown in Fig. 4.5.

[2] https://ckan.org/.

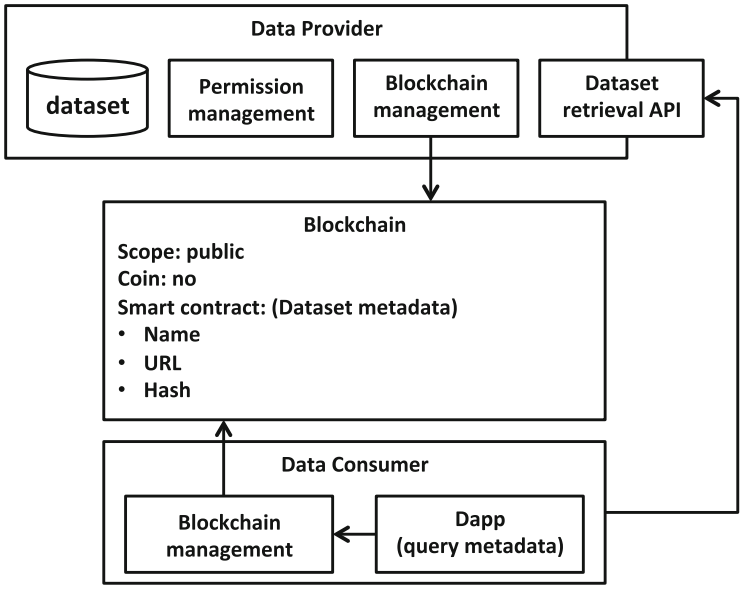


Fig. 4.5 Design for a registry using a public blockchain. © 2017 by the Commonwealth Scientific and Industrial Research Organisation, reprinted with permission

4.2.4 Non-functional Property Discussion

Integrity The conventional design relies on a registrar to create registry entries on behalf of data providers. New registry entries are validated by the registrar. In the blockchain-based design, registry entries can be created directly by the data providers, using their private key. Registry entries are validated by smart contracts checking data integrity conditions, and all transactions are validated by all processing nodes in the blockchain network. Data consumers hold a local copy of the blockchain, through which they access the registry.

Availability In the conventional design, the data registry system is a single point of failure for availability for all stakeholders. In the blockchain-based design, there is increased data redundancy which can improve read availability for data consumers. For the open data use case, write latency is not critical, which allows satisfactory service availability despite possibly lower write availability than the conventional design.

Interoperability In the conventional design, the datasets in different CKAN repositories refer to each other by importing the metadata from each other using standard formats but optionally with customer-defined fields. The blockchain-based design has a uniform technical infrastructure. The shared smart contract validation

rules will reduce the likelihood of incompatible data formats, which means different registries will be more consistent with each other.

Read Latency Reading in the conventional design is performed through a remote API over the Internet. Compared with the blockchain-based design, this is slower: a local blockchain node is collocated with the consumer's query interface, and reading is done locally at high speed.

Ease of Adding Providers In the conventional design, new data providers are added by the central registrar using registry backend services. In the blockchain-based design, new providers can join by independently creating a new public/private key pair. Authentication of their public key could be certified by a registrar on the blockchain or separately off-chain. Data providers must integrate with the blockchain, and should ideally run a blockchain node.

4.3 International Money Transfers

Many workers in Australia regularly send money back to their families overseas. These flows of cash constitute up to about 10% of GDP in some developing countries (and even 27% in Tonga and 20% in Samoa). Thus, high remittance costs have important implications on socio-economic development of these countries. Remittances are low-value, high-volume payments. However, remittance costs in Pacific Island countries are among the highest in the world. For example, to send $200 from Australia to Vanuatu costs $33.20 and $28.60 to Samoa.

There can be many parties involved in the chain of transactions made for these payments, and there is sometimes little transparency on the total cost of exchange rates and fees. Remittance payments can also be complicated by the difficulties of satisfying AML/CTF (Anti-Money Laundering/Counter-Terrorism Financing) regulation, especially where the receiving party may not have a bank account. These transactions can have high latency, with transaction times ranging from less than 1 h to 5 days.

In this use case, stakeholders include remitters, beneficiaries, and different types of financial institutions, including banks and Money Transfer Operators (MTOs). We consider the stakeholders and functions depicted in Fig. 4.6.

To be able to complete a remittance payment, both the remitter and beneficiary initiate a relationship with the financial institution. A Know Your Customer (KYC) process is conducted by the financial institution. The remitter pays a financial institution from the remitting territory who transfers the money across the border. Another financial institution from the beneficiary territory receives the money, exchanges it to local currency, and disburses it to the beneficiary. Prior to the completion of the exchange, and depending on the amount of money transferred, transactional level Anti-Money Laundering (AML) and Counter-Terrorism Financing (CTF) checks required by regulators in either territory (and in any financial

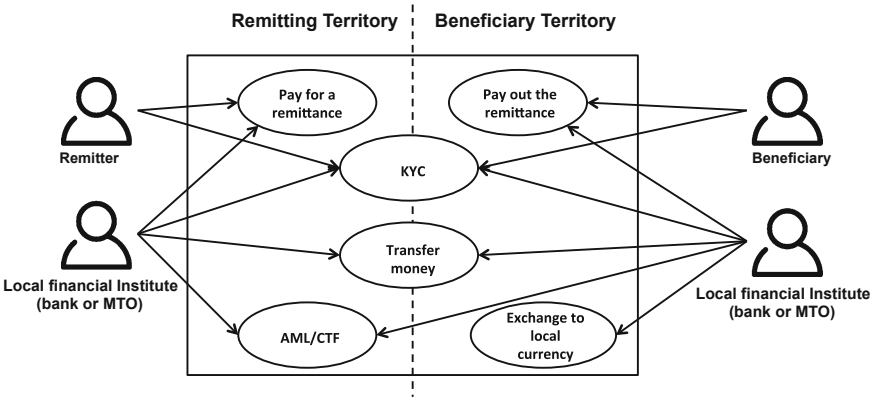


Fig. 4.6 Stakeholders and functions for remittance payments. © 2017 by the Commonwealth Scientific and Industrial Research Organisation, reprinted with permission

institutions in intermediate territories) may be performed on the identity of the remitter and beneficiary, perhaps including assessment of the purpose of the transfer.

4.3.1 Key Non-functional Requirements

- Transaction latency: completing a remittance payment should ideally be instantaneous, or at least take place comfortably within the context of human interaction with a physical kiosk or web form.
- Cost: the total cost of remittance should be a low percentage of the transaction value.
- Cost transparency: the total expected cost including fees and exchange rate should be visible to participants.
- Controlled confidentiality: for regulatory compliance, all required AML/CTF checks must be performed, but appropriate levels of commercial confidentiality must also be maintained.
- Barriers to entry: increased competition can drive lower costs and greater service innovation, but this requires low barriers to entry (cost, time, and regulatory burden) for new remittance service providers.

4.3.2 Conventional Technologies

The process for banks depicted in Fig. 4.7 starts when the remitter deposits money into their bank. The remitter's bank then initiates a SWIFT wire transfer to send the money across to the beneficiary bank, possibly through several intermediary

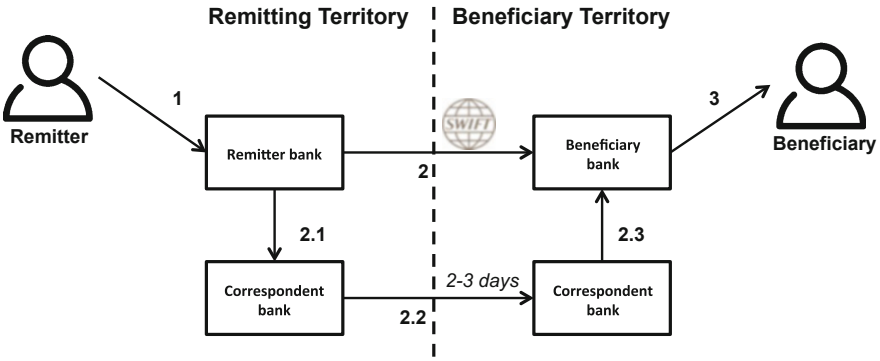


Fig. 4.7 Remittance through banks. © 2017 by the Commonwealth Scientific and Industrial Research Organisation, reprinted with permission

correspondent banks. It can take 2–3 days for the money to be sent. The receiving bank then informs the beneficiary's bank that the money in the foreign currency has arrived and transfers the local currency equivalent to the beneficiary's bank. Finally, the beneficiary's bank disburses the local currency to the beneficiary.

Another widely used way to do remittance is through a Money Transfer Operator (MTO), as depicted in Fig. 4.8. In this case, a remitter uses either cash or other payment instruments to pay the MTO. Once a group of payments is received, the remitting MTO pools all money into a single transaction. The MTO also prepares a file with instructions on breaking down the remittance to individual orders and sends the file to the beneficiary MTO. Then, the money is transferred by the MTO to its foreign bank as a normal international transfer, as per the above process. The bank

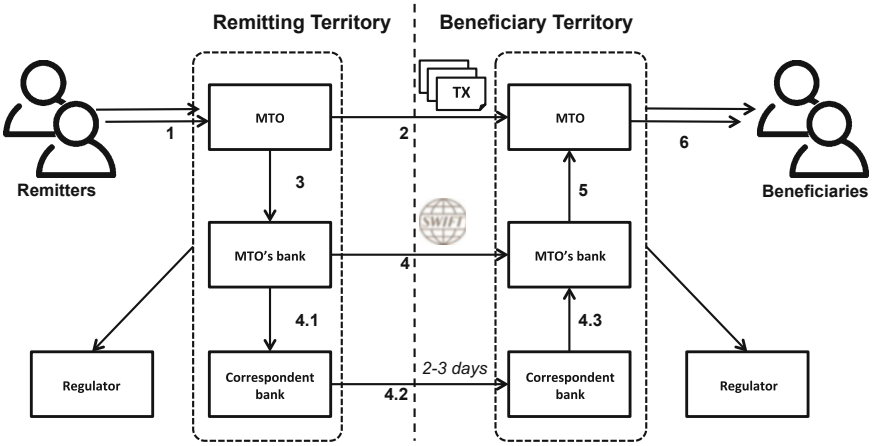


Fig. 4.8 Remittance through MTOs. © 2017 by the Commonwealth Scientific and Industrial Research Organisation, reprinted with permission

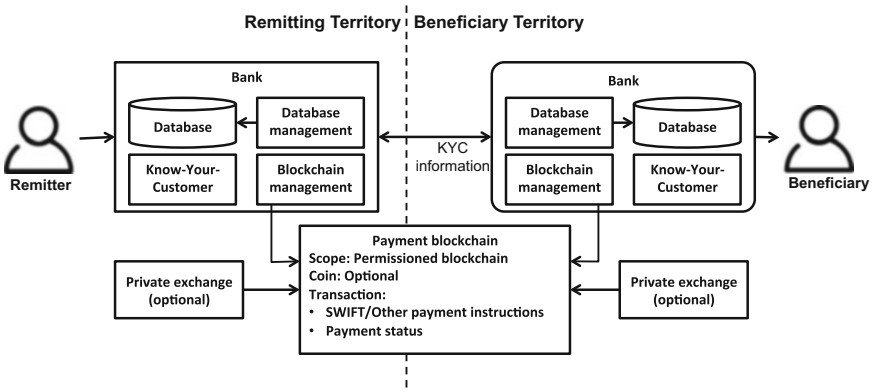


Fig. 4.9 Payment through blockchain. © 2017 by the Commonwealth Scientific and Industrial Research Organisation, reprinted with permission

charges the MTO once for all the remittances. When the beneficiary MTO receives the money, it distributes it according to the instructions received earlier.

4.3.3 A Blockchain Solution

Banks, financial institutions, and MTOs could join a private blockchain to enable real-time settlement, as depicted in Fig. 4.9. Apart from speeding up money transfers, blockchain could also help banks to operate continuously, 24 h a day. The on-chain portion of the design can include SWIFT instructions or other payment instructions and the payment status. The native currency of the blockchain can be used as an intermediary currency by banks to facilitate foreign exchange. KYC and risk information, fees, and foreign exchange rates are exchanged through conventional means, off-chain.

When Bitcoin is used, this is sometimes called 'rebittance'. Some companies use Bitcoin directly as an intermediary currency for foreign exchange. The underlying Bitcoin layer is invisible to end users. In this case, every remittance has a corresponding transaction recorded on the Bitcoin blockchain. Other companies maintain a separate blockchain to facilitate settlement among branches and anchor their blockchain with the Bitcoin blockchain as a way to leverage Bitcoin's immutable, independently auditable ledger.

4.3.4 Non-functional Property Discussion

Transaction Latency The systems following conventional designs can result in time-consuming transactions, e.g. depending on the route, specifically the number of

correspondent banks involved. End-of-day batch processing causes delays of up to 24 h, and time zone differences can cause delays of up to 24 h. The blockchain-based design enables real-time processing with latencies that vary from seconds to hours, depending on the blockchain. For example, on Bitcoin the latency averages around 1 h if 6 confirmation blocks are used; using public Ethereum with 12 confirmation blocks would on average take around 3 min.

Cost In both designs, remitting banks charge transaction fees, and liquidity providers charge via the spread on foreign exchange (FX) rates. There are also correspondent bank fees in the conventional design.

Transparency In the conventional design, each bank in the payment chain is aware of its own actions, but some KYC information is transmitted through the chain of correspondent banks. How FX spread is calculated and what will be charged in fees is not always predictable. In the blockchain-based design, a common shared view of the payment status enables real-time fraud analysis and prevention. On Bitcoin, regulators and others can access historical data in the blockchain but would need additional information to know how to interpret the pseudonymous addresses and the identities of senders and recipients.

Controlled Confidentiality In the conventional design, KYC regulatory compliance requires costly technology capabilities and complex business processes. There is substantial duplicated effort between banks and other financial institutions. The blockchain-based design replaces intermediary banks with a blockchain to provide a shared record of payments and KYC checks and thus may simplify regulatory compliance along the payment chain. Some automated and real-time compliance checks may be available on-chain using smart contracts, depending on the blockchains used.

Barriers to Entry The conventional design requires participants to have banking or financial services licenses, and business relationships with correspondent banks. The second design requires new technology development and integrations, but some existing transaction standards can be reused. Interaction between separate proprietary blockchains would require inter-ledger protocols. Public blockchains have low barriers to entry for new participants, but regulatory or banking constraints for digital currency exchanges apply to end-points within countries.

4.4 Electricity Contract Selection and Continuous Reporting

Electricity consumers may change their electricity retailers based on their usage and current offers from electricity retailers. Typically, there are conditions associated with the contract between the electricity consumer and the retailer. For example, retailers may offer discounts if bills are paid on time or may require exit fees if the contract is terminated ahead of time. Some retailers also allow flexible payments, such as weekly, fortnightly, or monthly payments. There are two participants in this

scenario: the end user and the electricity retailer. We assume that a smart meter is attached to the end user's place of supply and that this smart meter is connected to the network and can digitally sign messages using a private key.

4.4.1 Key Non-functional Requirements

- Integrity: The monthly usage of an electricity consumer is an important criterion for consumers to select an electricity retailer and for electricity retailers to make special offers. Therefore, accurate records of usage are important to prevent deception between the parties.
- Privacy: Current and historic electricity usage data can be used to infer private information—researchers have even shown that accurate high-frequency smart meter readings allow identifying which movie an end user is currently watching. More coarse-grained data could also be used by burglars to find out when someone is on vacation. Therefore, usage data should only be shared at the discretion of the end user.
- Transparency: Historical electricity usage, perhaps associated with previous electricity retailers, could be used by a prospective retailer to customize new special offers. As discussed above, we assume that consumers are able to authorize the sharing of their usage information with other parties.

4.4.2 Conventional Technologies

In conventional environment, every electricity retailer uses its own bespoke system to maintain customer data and smart meter information. Historical electricity usage is not normally shared among electricity retailers. Payments are made through traditional banking systems.

4.4.3 A Blockchain Solution

In this blockchain-based design, we propose using a consortium blockchain as a platform to track historical electricity usage of every smart meter and to provide payment services. The architecture of the solution is shown in Fig. 4.10.

When a user wants to find a new retail supplier, they create a retailer selection smart contract on the blockchain, against which retailers can bid. To bid, retailers create a smart contract offer from an offer template. The offer contract is defined using variables such as start time, end time, energy level, level price, service fee, and charge date. Transactions listed on the blockchain provide a history of usage associated with smart meters and users. This information can be accessed by retailers

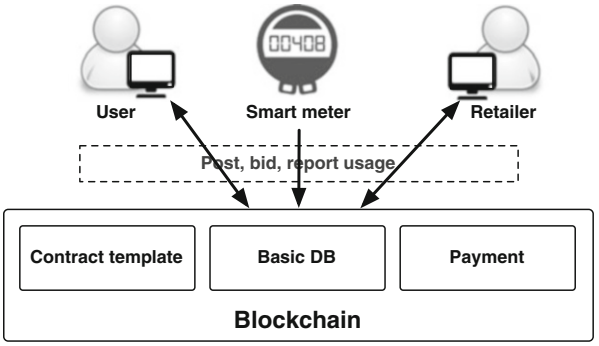


Fig. 4.10 Architecture of blockchain-based electricity contracts using smart meters. © 2017 by the Commonwealth Scientific and Industrial Research Organisation, reprinted with permission

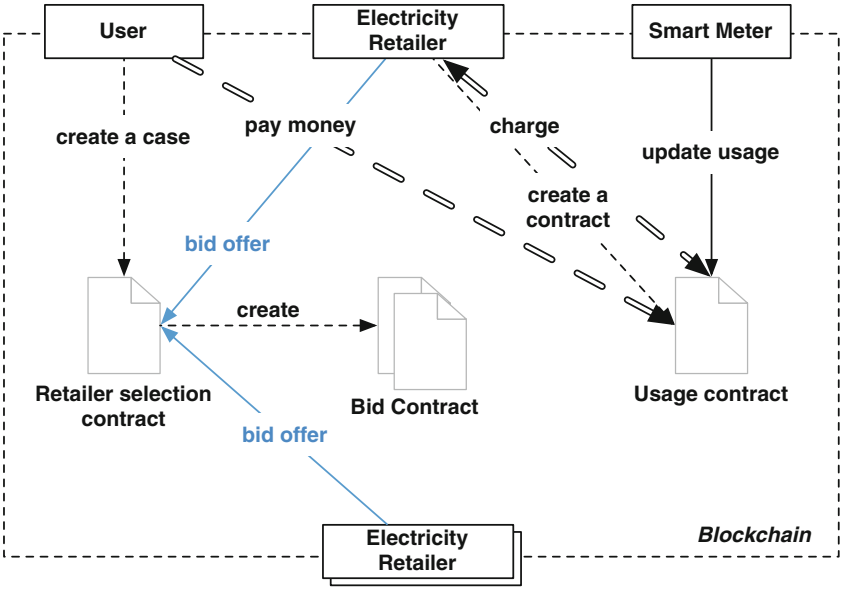


Fig. 4.11 Interaction of smart contracts among themselves and with other entities. © 2017 by the Commonwealth Scientific and Industrial Research Organisation, reprinted with permission

as they prepare their offer. The user's retailer selection contract collates all the bids, which can then be shown on a web page accessible to the user for final selection. The interaction with and among the relevant smart contracts is shown in Fig. 4.11.

After the electricity retailer is selected, a usage contract that is specific to the pair of the user and the retailer is generated and uploaded to the blockchain. The usage contract is used by the smart meter to report the monthly usage. There are two options to pay the bills. The user could either pay the bill to the retailer's

account directly before the deadline or deposit money into the contract and let the retailer withdraw from it when the payment is due. When the user decides to switch retailers, she could create a replacement usage contract from the new retailer selection contract. The usage contract with the previous retailer will terminate after the new contract is created.

4.4.4 Non-functional Property Discussion

Integrity The conventional approach relies on individual electricity retailers to maintain the internal system and usage data. New usage data is validated solely by the electricity retailer. In the blockchain-based design, usage records can only be created by the smart meters using their private keys. All transactions are validated by all processing nodes in the blockchain network. Electricity retailers hold local copies of the blockchain, through which they can access the historical electricity usage of any smart meter.

Privacy In the conventional design, the usage data is only shared with the current electricity provider. In contrast, in the blockchain-based design, the data is shared with all electricity providers that are on the consortium blockchain. Design of the blockchain-based system is important to meet privacy requirements. For example, the blockchain might be a private permissioned blockchain or distributed ledger, and users and smart meters might access the blockchain only through controlled web interfaces or APIs.

Transparency In the conventional design, information from smart meters and historical usage are stored by each separate electricity retailer. Such information is not accessible by other electricity retailers. The consortium blockchain used in the blockchain-based design provides a common shared data storage for historical usage associated with any electricity retailer.

4.5 Further Reading

This chapter is partly based on our earlier works (Staples et al. 2017).

A detailed use case from a startup company, focussing on the reduction of counterparty risks in agricultural supply chains, is described in Chapter 12.

A model-driven approach is proposed in Weber et al. (2016), which can generate smart contracts automatically from process models. This approach is also discussed in Chapter 8.

Reports from the World Bank (Ratha et al. 2016; The World Bank 2016) provide data and insights about remittance.

Bitcoin has been applied to smart meters deployed in South Africa (Prisco 2015), where each smart meter is equipped with its own Bitcoin address. The smart meters

directly pay for their metered electricity and water supply from their balance and form an interesting middle ground between prepaid and post-paid services. Bitcoin has also been applied in smart grid scenarios (Dimitriou and Karame 2013) to facilitate aggregating energy production and consumption reports without relying on a single point of trust. This enables anonymous tasking and privacy-preserving billing and barter of energy.

The use of smart contracts to enable machine-to-machine communication in IoT has, for example, been demonstrated by the ADEPT (Autonomous Decentralized Peer-To-Peer Telemetry) project (IBM 2015).

Part II
Architecting Blockchain-Based Applications

Chapter 5
Blockchain in Software Architecture

Software components are the fundamental building blocks for software architecture. In a blockchain-based system, a blockchain platform is a component. A reference architecture for a software system where blockchain is one of the components is shown in Fig. 5.1. Viewing the blockchain as a software component helps us understand important architectural impacts it has on the performance and quality attributes of systems. These attributes can include security, privacy, scalability, sustainability, and more. We can then consider design trade-offs regarding these quality attributes and provide rationales to support architectural decisions about whether to employ a blockchain or some conventional component.

5.1 Blockchain as an Architectural Element

As a component, blockchain has unique properties and limitations. Blockchains are complex, network-based software components, which can provide data storage, computation services, and communication services. Blockchain features can include cryptographically secure payment, mining, transaction validation, incentive mechanisms, and permission management. A so-called oracle may supply information about the external world to the blockchain, usually by adding that information to the blockchain as data in a transaction.

One of the main kinds of architectural decisions is about which pieces of functionality should be allocated to which components. *For blockchain-based systems, this includes the key decisions about which parts of the data and computation should be placed on-chain or kept off-chain.* Part of an application can be implemented inside the blockchain component using the blockchain ledger and smart contracts. However the amount of computational power, data storage space, and control of read accesses on a blockchain can be limited. So, parts of an application implemented outside the blockchain component might host off-line data and application logic.

X. Xu et al., *Architecture for Blockchain Applications*,
https://doi.org/10.1007/978-3-030-03035-3_5

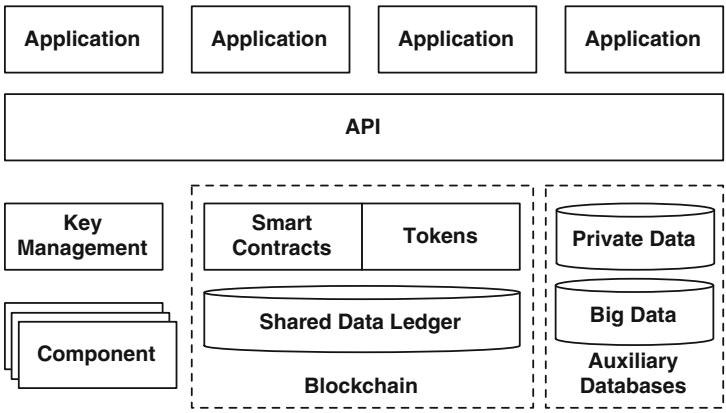


Fig. 5.1 Blockchain in a software architecture. This work is based on an earlier work: Xu et al. (2018) © ACM, 2018. https://doi.org/10.1145/doi. Included here by permission

Blockchain transactions and their effects sit at the interface between on-chain and off-chain functions. Blockchains can be used as software components, which can provide data storage, computation services, communication services, and asset management and control functions. We discuss these aspects in the remainder of this chapter.

5.2 Blockchain as Storage Element

Blockchains emerged as the key technology behind Bitcoin. The Bitcoin blockchain is a public ledger maintained by all the nodes within its network and stores all transactions that have ever occurred in the system. Later, the technology concept was generalized to a distributed ledger able to verify and store a wider variety of transactions, including transactions that do not transfer cryptocurrency.

As a data structure, a blockchain is an ordered list of blocks, where each block contains a small (possibly empty) list of transactions. Each block in the data structure is 'chained' back to the previous block, by containing a cryptographic hash of the representation of the previous block. Historical transactions in the blockchain may not be deleted or altered without invalidating this chain of hashes. Combined with computational constraints and incentive schemes on the creation of blocks, this can in practice prevent tampering and revision of information stored in the blockchain.

Transactions on a blockchain represent authorized state transitions. Transactions can record data and transfer control of digital assets among participants. Cryptocurrencies are one kind of digital asset, but other kinds of digital asset tokens can be implemented on blockchains. For example, on Ethereum digital asset tokens can be represented using smart contracts, which can represent the tokenized asset

and store holdings as values of private variables. Public key cryptography and digital signatures are normally used to identify accounts and to ensure integrity and authorization of transactions initiated on a blockchain.

There are two ways to store data on the blockchain. One is to add data into transactions, which is the only option in Bitcoin; the other is to add data into contract storage, which can be done e.g. on Ethereum. Both ways store data through submitting transactions to the blockchain, which may contain the information of money transfer (possibly with a transfer value of 0), together with optional other data. After the transaction is included in the blockchain, the data becomes publicly accessible to all the participants within the network.

There are various representations of cryptocurrency holdings. In Bitcoin, the holdings of an address comprise the collection of unspent transaction outputs (UTXO) from all previous transactions to that address. In Ethereum, the holdings of an address are represented in a global system state. In Ethereum, every smart contract has its own storage which only it can update. Contract storage can be viewed as a flexible key-value data store. Smart contracts have an address, which can be used to invoke the contract. Blockchains are immutable, so the updates to variables in the contract store do not change the data in old blocks. Instead, the transactions only update the values of those variables in the contract store.

5.2.1 Comparison with Centralized Databases

Operation Shared data stores, like key-value stores, provide a basic Create/Read-/Update/Delete (CRUD) interface. A blockchain is an append-only data store and does not support in-place updates but rather only supports the creation of new transactions. The current view of smart contract variable values can mimic the behaviour of conventional data stores. However, any changes/updates on contract states are appended to the blockchain as new transactions. An analogy with this so-called ledger in data stores is the concept of log where data items get appended but never deleted or updated. This immutability-of-stored-information property is key to the traceability of assets recorded on blockchains.

Consensus Protocol Traditional shared data stores have their own consensus protocols to synchronize replicas in a fully trusted environment, such as 2-Phase Commit and Paxos. Blockchains also rely on consensus protocols, and private blockchains often use the same protocols as traditional shared data stores. However on public blockchains, the assumptions required for conventional consensus protocols do not hold. In particular, on a public blockchain there may be no master nodes, many thousands of nodes, and there may be an unknown number of nodes. Each of these violates assumptions required for some conventional consensus algorithms. Some private blockchains assume that nodes are somewhat trustworthy, but on a public blockchain that assumption is not always reasonable. A comparison of consensus protocols used for blockchains and for general distributed systems is given in Section 5.2.4.

Consistency Blockchains validate the consistency of transactions by using global rules implemented in the blockchain platform, and by using application-specific rules implemented in smart contracts. Global rules on public blockchains include, for example, that regular cryptocurrency transactions do not create money nor transfer money without authorization.

5.2.2 Comparison with Cloud Services

When keeping data on conventional cloud storage, cloud providers are trusted to store data uploaded by users and to provide access to that data. Users are normally not able to influence to how the data is stored. Data integrity and access availability may not be guaranteed.

In contrast, on blockchain there is no need to trust a single entity. Storage integrity is guaranteed (probabilistically) through the actions of the collective of nodes that operates the blockchain. Users can monitor that collective and could even participate themselves as nodes that store the blockchain if desired. A variety of on-premise computers or independent cloud providers could be used to operate nodes. At the application level, users as smart contract developers define their own storage mechanisms and interaction with off-chain services. While high read availability can be achieved, by reading from multiple independent blockchain nodes, there are no guarantees or defined service-level agreements (SLAs) provided by public blockchains.

5.2.3 Comparison with Peer-to-Peer Data Storage

Peer-to-peer technology can be used for distributed data storage and file sharing. Such systems allow users to access data that is stored in other computers connected to the same peer-to-peer network. A centralized server is not required. Existing platforms include BitTorrent[1] and IPFS (InterPlanetary File System).[2] These peer-to-peer systems use various mechanisms to share data with peers and to replicate data across nodes. IPFS is an open-source content-addressable, globally distributed file system for sharing a large volume of data with high throughput.

In contrast, on blockchain access to data will be possible while users have access to nodes that are active in the blockchain network. Users could access any node in the network collective, because all nodes will have the same shared copy of the blockchain data. This can give very high levels of availability for blockchain-based data storage. On the other hand, blockchain is not suitable for storing large data, and

[1] http://www.bittorrent.com/.
[2] https://ipfs.io/.

so blockchain is often combined with other data storage mechanisms, such as peer-to-peer data storage. Blockchain can then provide integrity in the sense of revealing possible tampering of off-chain files. However, blockchain could not stop off-chain manipulation of files or file shards, only make it detectable.

5.2.4 Comparison with Replicated State Machines

Replicated state machines are a technology that is somewhat similar to blockchains. We discuss the differences in terms of key properties below.

Fault Tolerance Replicated state machines are a mechanism to implement fault-tolerant services in a distributed system. To cope with failures, they replicate state at several servers and coordinate service requests issued by the clients. Similarly, the blockchain uses distribution to not depend on any single entity. Smart contracts on blockchain can then implement many kinds of service logic.

Consensus Replicated state machines typically rely on a consensus protocol that takes as input update requests from components and decides upon receiving these requests. In the case of a distributed locking service, the consensus will guarantee that only one particular client acquires a lock, even if multiple clients request it concurrently. Blockchain systems also use a consensus protocol to ensure that among multiple conflicting proposed transactions, only one is approved for inclusion in the blockchain.

Voting To reach consensus on a transaction request, replicated state machines typically require a quorum of voters and may use a concept of weighted votes. Typical blockchain implementations also require a large enough portion of the community operating the system to agree to achieve consensus. In Ripple, this is when a minimum of nodes in a unique node list have voted, whereas in Bitcoin this is (tentatively) when a sufficiently long chain of blocks (ratified by others) is discovered.

Communication A replicated state machine supports communication by transmitting state update data among components. Components can store and retrieve information that will persist despite failures. The system guarantees that information stored by one component is replicated and delivered to the other components even when some failures occur. Public blockchains offer no strict guarantees on transaction inclusion, but once transactions are seen as committed, they have been replicated and persisted and are exceedingly unlikely to be removed.

Cryptography To address arbitrary failures or Byzantine failures, replicated state machines exploit security mechanisms. The sender of a message is typically authenticated with public key cryptography by signing their messages with a private key. Digital signatures are similarly used by blockchains to authorize transactions.

Facilitation Finally, a replicated state machine totally orders the requests from components. It controls concurrency by scheduling requests issued by components and thus serves as a facilitation connector. Such a total order is also a key property of the blockchain: blocks in the blockchain data structure are totally ordered and so are the transactions within a block.

5.3 Blockchain as Computational Element

In first-generation blockchains like Bitcoin, there was very limited native capability for programmable transactions. Native smart contracts on Bitcoin are very simple and do not support complex control flow. Some external services attempted to address this, to allow end users to build self-executing contracts on the Bitcoin blockchain network,[3] but the blockchain platform does not guarantee the integrity of the execution of these smart contracts. Instead, smart contract execution is performed by external oracles.

Ethereum is the most widely used blockchain allowing smart contracts to be written in a Turing complete language that is in principle as expressive as every other general purpose programming language. Ethereum can be seen as general computational platform, albeit currently with severe practical limitations on computational complexity. This kind of capability significantly expands the power of blockchain systems and increases their range of use and potential for innovation.

In Ethereum, smart contracts are a first-class element. They can control cryptocurrency and express triggers, conditions, or business logic (see also Chapter 8), to enable complex programmable transactions. Smart contracts are used by components connected to a blockchain to reach agreements and solve common problems with minimal trust. A common simple example of a smart contract-enabled service is escrow, which can hold funds until the obligations defined in the smart contract have been fulfilled. Smart contracts can also be used to enable machine-to-machine communication in IoT, for example, as demonstrated by IBM's ADEPT (Autonomous Decentralized Peer-To-Peer Telemetry) project.

The status of smart contracts as legal contracts is currently debated. A legal contract is an agreement between parties, and a computer program is either the text of source code or an executing physical machine. Therefore smart contracts, as computer programs, may be the wrong category of thing to be a legal contract. Nonetheless, a smart contract may provide evidence for there being a legal contract and may be able to facilitate the execution of a legal contract. Importantly, as a mechanism for the execution of provisions of a legal contract, smart contracts can carry and conditionally transfer digital currency and other digital assets or tokens between parties. This can be done in a predictable and transparent way on the neutral ground provided by the mechanized infrastructure of a blockchain.

[3] http://www.smartcontract.com/.

5.4 Blockchain as Communication Mechanism

Software components communicate by using communication elements, which are also components. A communication element can transfer data and coordinate computation among components. Blockchain systems perform all of these functions as well, but of course with some differences to traditional communication elements.

5.4.1 Data Communication

Components can use blockchain as a mediator to transfer data, as shown in Fig. 5.2. The components at the application layer exchange data by sending data to the blockchain using transactions, and query the blockchain data structure to retrieve the data.

Most blockchain platforms provide an API or tools to access and filter the historical transactions. Ethereum suggests to cache all transactions to prevent the blockchain network from being stressed by frequent queries.

An alternative, discussed in some detail in Chapter 8, is to have a component that continuously monitors the updates from new blocks. This component can store relevant data in a local database or pass it on through proactive API calls to components on the application layer.

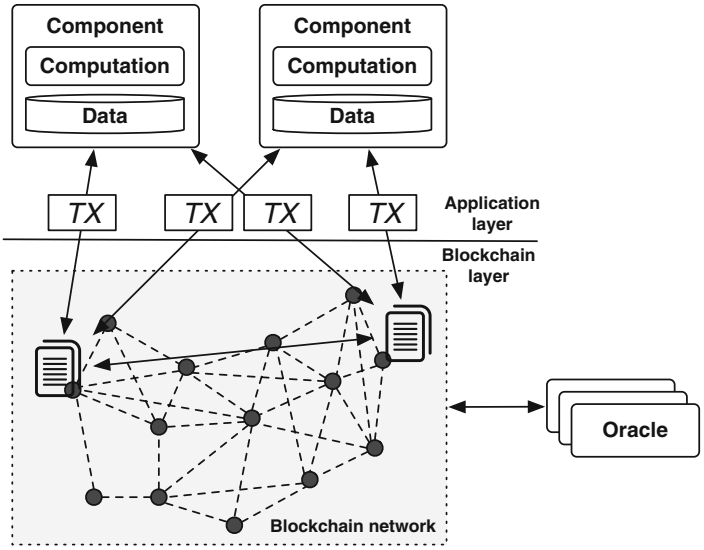


Fig. 5.2 Interaction between applications and blockchain. © 2016 IEEE. Reprinted, with permission, from Xu et al. (2016)

5.4.2 Computation Communication

Different components of an architecture can coordinate computation through a blockchain. To do so, it is possible to submit transactions to smart contracts to invoke their functions or use an oracle to sign transactions that depend on external state.

Typically the control flow of an application is initiated from externally owned accounts and is transferred among contract accounts. Smart contracts behave like agents that live in the execution environment of the blockchain system and its network. Contracts are instantiated by submitting transactions with the code of the contracts to the blockchain network—depending on the blockchain platform, the code is source code or compiled. A smart contract defines a set of functions. When invoked in a transaction, the contract runs the function code using the supplied parameter values. Contracts can also create new contracts, and can terminate themselves. A contract cannot respond to transactions after termination, but the code of the contract remains on the blockchain, permanently stored in the transaction that created the contract.

The execution environment of a blockchain system is a closed environment, which is not allowed to import external states through polling external servers. To address this limitation, *oracles* evaluate conditions about the external world which cannot be derived solely from information within the blockchain. An oracle facilitates component coordination with external state. An oracle is a component in a blockchain-based system. Some blockchain platforms provide direct support for oracles, while in other blockchain platforms, an oracle is an independent external service that interacts with the blockchain through normal transactions. When validation of a transaction depends on some external state, platform-supported oracles can validate and sign the transaction. This may block progress of the transaction until the oracle completes. Oracles that are external services inject data into the blockchain by adding a transaction, and other smart contracts can then use that data to validate transactions. This can reduce the above-mentioned delay but can increase the delay between the external state changes and the time when those are recorded in the blockchain. Often oracles are automated external systems, but sometimes an oracle represents decisions made by a human, e.g. an arbitrator. When automated external service oracles are used, they can periodically update values. Regardless of the approach, oracles are a trusted third-party. However, this does not always introduce additional trusted parties, if they are already trusted parties. For example, for government services, the government is inevitably a trusted party.

5.5 Blockchain as an Asset Management and Control Mechanism

Blockchains can be used for asset management by using the concept of tokenization. Tokens can represent either digital assets or physical assets. Such assets can be fungible or non-fungible. *Fungible* assets are interchangeable, for example,

cryptocurrencies, gasoline, and commodities. *Non-fungible* assets are often unique and cannot be interchanged, for example, CryptoKitties,[4] artwork, and land.

On the first generation of blockchains, the cryptocurrency is the native asset. However, the identity of portions of that cryptocurrency or other associated data can be used to represent other kinds of assets. Such tokens are generally used to track claims of title over physical assets. Transactions on blockchains record the transfer of title from one user to another in the system. The Bitcoin blockchain allows developers to add 40 bytes of arbitrary data to a transaction. This has been used to implement other cryptocurrencies or tokens as overlay networks on Bitcoin. For example, colored coins[5] 'taint' a subset of Bitcoins to represent and manage real-world assets. However, using the native cryptocurrency token in a blockchain for tokenization of other assets is limited, because few attributes can be recorded and few conditions can be checked within the blockchain.

Second-generation blockchains, like Ethereum, provide more expressive data structures and smart contracts. This provides more flexibility for tokenizing a wider variety of assets. Tokenization as a process starts when an asset under custody is represented using a cryptographic token. The control of this token aligns with the ownership of the corresponding asset. The reverse process can take place if the user redeems the token to recover the asset. By using smart contracts, some conditions can be implemented and associated with the transfer of ownership. ERC20[6] has been proposed as a standard for Ethereum-based fungible tokens. ERC721[7] is currently a draft standard for Ethereum-based non-fungible tokens. These token standards describe the functions and events that token smart contracts should implement. Newly proposed tokens should follow the respective standard.

Note that title over assets is a legal construct, which might not always completely align with the records on a blockchain. For example, the ownership of assets might be legally transferred during bankruptcy proceedings, without recourse to the blockchain. So, unless the blockchain is backed by legislation (similar to Torrens title legislation for land) as an authoritative title of register, it will not necessarily be authoritative.

5.6 Integrating Blockchain into a System as a Component

In the system shown in Fig. 5.1, the blockchain stores and shares data and executes smart contracts. The blockchain component might also control digital currency or represent other assets. Due to limitations of privacy and scalability, there are also off-chain auxiliary databases used in the system. First, private data is stored in an

[4] https://www.cryptokitties.co/.

[5] http://coloredcoins.org/.

[6] https://theethereum.wiki/w/index.php/ERC20_Token_Standard.

[7] https://github.com/ethereum/EIPs/issues/721.

internal database. Second, large data is stored separately, e.g. in a cloud service. There is an API layer between the three data storage mechanisms. Key management is an essential component when working with blockchains. Every participant in a blockchain network has one or more private keys, which are used by the participant to digitally sign its transactions. The security of these private keys is very important. If the private key of a user is stolen, any other user holding it can forge transactions from that user to spend the assets belonging to the user or to invoke smart contracts in their name.

5.7 Summary

In this chapter, we characterized blockchain functions from software architecture perspective and describe blockchain in the role of a software component. Blockchain can be used as a storage element, a computation element, or a communication element for interaction between system components. It can also be used as an asset management and control mechanism. We compared blockchain as a software element with central shared data stores, cloud storage, peer-to-peer storage, and replicated state machines.

5.8 Further Reading

This chapter is partly based on our earlier works (Xu et al. 2016; Yu et al. 2017).

The concept of software components and software connectors was introduced by Clements et al. (2003). A technical survey on blockchains and distributed ledgers is given in Tschorsch and Scheuermann (2016).

In this chapter, blockchain is compared with centralized databases having their own consensus protocols to synchronize replicas in a fully trusted environment, such as 2-Phase Commit and Paxos. More details are discussed in Kemme and Alonso (2010). Blockchain is then compared with cloud services. Cloud providers have access to the data of their users. The issues of data privacy and provenance on cloud are discussed in Ion et al. (2011) and Asghar et al. (2012). Blockchain is further compared with replicated state machines (see Schneider 1990; Lamport 1998). Replicated state machines aim to address arbitrary failures or Byzantine failures, which are described in Lamport et al. (1982) and Castro and Liskov (1999). The voting mechanism used by replicated state machines is discussed in Malkhi and Reiter (1997) and Gifford (1979).

Chapter 6
Design Process for Applications on Blockchain

with Sin Kuang Lo

Software design is a creative process, which includes proposing and evaluating solutions to complex problems with many conflicting constraints. The final design of a software system is the result of many design choices about the selection, configuration, and integration of software, hardware, and communications components. This chapter presents a design process for architecting systems based on blockchains.

For a system that can potentially use blockchain, the first design choice is to decide whether to use a blockchain or conventional technologies. We discuss this choice in Section 6.1 and give four examples in Section 6.2. When using a blockchain, there are subsidiary design choices including whether to use a private blockchain or a public blockchain, what consensus protocol fits best, and what the block frequency should be. Chapter 3 identifies a variety of design choices, and in Section 6.3 we discuss how to address them. Often in a blockchain-based system, some data is stored on the blockchain, while other data is stored and communicated using conventional technologies, so another design choice is which data should be stored where.

6.1 Evaluation of Suitability

Due to their fundamental properties and limitations, blockchains do not fit all scenarios. Thus, before designing a system, the suitability of blockchain needs to be evaluated against the scenarios and requirements.

Figure 6.1 shows a process to evaluate the suitability of blockchain technology. There are seven main questions to be answered, shown as white diamonds. For some of them, subsidiary questions are shown as grey diamonds. The following subsections discuss these questions in detail.

X. Xu et al., *Architecture for Blockchain Applications*,
https://doi.org/10.1007/978-3-030-03035-3_6

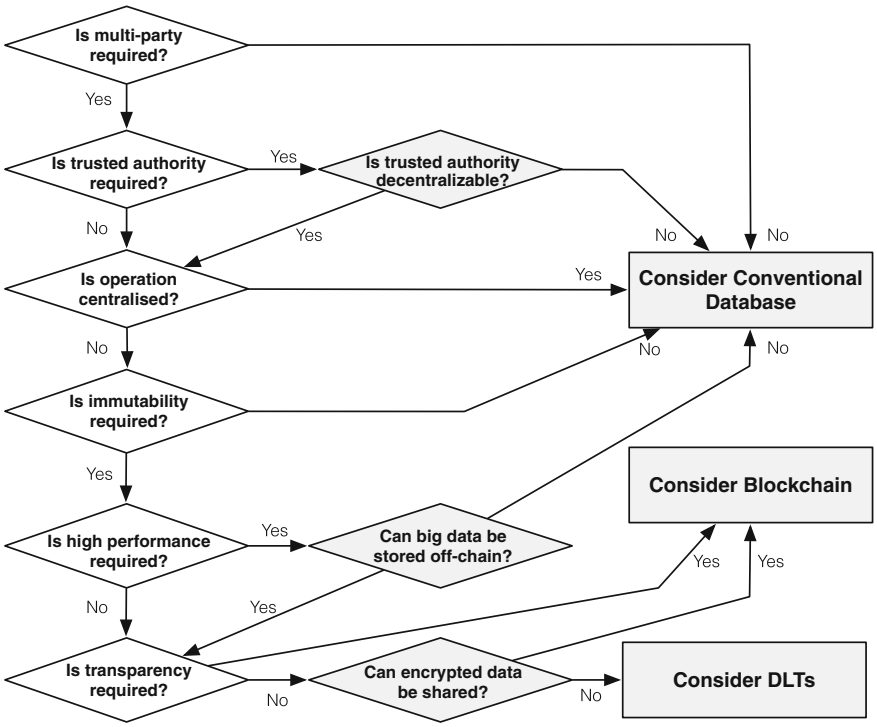


Fig. 6.1 Evaluation of suitability of blockchain and other DLTs. © 2017 IEEE. Reprinted, with permission, from Lo et al. (2017)

6.1.1 Multiparty

Does the system need to serve multiple different parties? A blockchain is not suitable for systems that only serve individual isolated users, because a conventional database will be simpler and more efficient. There are many different kinds of multiparty systems. Consider the supply chain domain, which has complex, dynamic, multiparty arrangements with regulatory and logistical constraints spanning jurisdictional boundaries. Information exchange in a supply chain can be as important and difficult as the physical exchange of goods. The multiple users here may be manufacturers, shipping companies, transport infrastructure organizations, financial services firms, or regulators. Another example domain might be inter-bank payments and reconciliation. Here the multiple parties are at least two different banks, but may also include the account holders performing payment transfers between the banks. So, parties might be organizations or individuals. In these examples, the different parties are legally distinct. However, even within one large enterprise (or government), there may be different functional or geographic divisions or departments. These informational or administrative 'silos' may need to

be served as multiple parties. Blockchains can be suitable for supporting multiparty systems, because the blockchain is a physically distributed but logically centralized infrastructure, providing a single view of truth across those parties.

6.1.2 Trusted Authority

A trusted authority is an entity that is relied upon to perform a function, like operating a system. If a single party can or must be relied upon as a trusted authority by all of the parties served by a system, then a blockchain may not be necessary. Instead, that trusted authority could implement a traditional centralized solution using conventional technologies. Most current complex systems are controlled by a trusted authority. Examples of these authorities include banks and government departments. The scope of the system being designed is important in deciding this question. For bank accounts, the bank will be a trusted authority. However, for inter-bank payments, each participating bank will not be a trusted authority; instead the conventional approach is for banks to collectively rely upon separate authorities to facilitate inter-bank payments. For example, within a country that trusted authority might be a central bank.

Relying on a trusted authority creates a single point of failure for the system. When a trusted authority experiences a problem, users accessing its services are affected. Technical single points of failure can be mitigated by using redundancy in conventional distributed systems architectures. However, those solutions do not address single points of organizational or business failure that remain present when relying on a trusted authority. These possible failures might include business failures, service interruptions, data loss, or fraud. For situations where the trusted authority is a monopoly or oligopoly service provider, there is also the possibility of what economists call 'rent-seeking' behaviour, which can unreasonably limit access to the service and can reduce efficiency through excessive charges.

Even when a natural trusted authority might in principle be available, in practice it might be difficult for everyone to accept reliance on that party. Consider a government with multiple different departments or agencies. Large enterprises or government could in principle define a central agency to provide services for coordinated operation across their whole organization. However, centralization of services can be perceived as a loss of control or power, and so in practice it may be difficult to achieve this kind of administrative centralization.

Blockchain can support systems where there is no single party that is acceptable or suitable for operating the system. That is because a blockchain is operated jointly by a collective of nodes. Using a blockchain does not remove trust, because users are still exposed to risk in their use of blockchain technology. In a blockchain, what is trusted (i.e. relied upon) is the blockchain software, the incentive or contractual mechanisms driving the behaviour of processing nodes that operate the blockchain system, and the trusted third-parties that act as 'oracles' which record information about the external world on the blockchain. Although a blockchain does not remove

trust, it can remove the need to trust a single specific third-party to maintain a ledger and so is sometimes called a 'distributed trust' mechanism.

6.1.3 Operation

Given that a system supports multiple parties, and given no party is suitable as a trusted authority for administering the system, might centralized operation of the system still be possible? A common approach is that a group of parties might form a joint venture to operate a conventional centralized system. Credit card associations such as Visa and Mastercard are examples of this approach, formed as kinds of joint ventures between banks.

However in some cases, it is not possible or desirable to centralize operation of the system. The centralized operation of the system may lead to the administering party becoming a trusted authority, which will not always be acceptable to the parties using the system. Forming a new entity like a joint venture might be too costly for a given scenario. Also, the centralized administration of the system may still allow single points of business failure for the system. A distinctive benefit of blockchain-based systems is that there does not have to be a single authority or system operator. Eliminating single points of failure can increase system reliability or availability.

6.1.4 Data Immutability and Non-repudiation

Is data immutability required and acceptable? *Data immutability* means data cannot be changed or altered after its creation. Immutability supports *non-repudiation* which is the assurance that a party cannot deny the authenticity of their signature on a document or a message from them. Blockchains naturally support data immutability in the ledger, whereas conventional technologies naturally support mutable data. What is important as a requirement can vary from system to system.

Although the blockchain transaction history is immutable, the latest view of the current state in a blockchain can change. For example, a transaction may need to update the owner of an asset. What is recorded to the ledger in this case is the new owner for the asset, and so all that changes is our view of the latest owner. In a blockchain, the linking of blocks in a chain of cryptographic hashes supports immutability for historical transactions. In practice, past blocks in the blockchain data structure cannot be changed because it is continually replicated across many different locations and organizations; attempts to change it in one location will be interpreted as an attack on integrity by other participants and will be rejected. In economies where third-party service providers are not always trustworthy, a significant benefit of blockchain systems may be in the strong support that they can provide for immutability and non-repudiation. On a blockchain, the immutability

of historical transactions which are cryptographically signed means that there is always strong evidence that those transactions were performed by someone with control over those cryptographic keys.

On the other hand, it is not possible to change the transaction history in most blockchains. This is normally a good thing in supporting data integrity. However, it can cause problems if blockchain contains illegal content, or if a court orders content to be removed from the blockchain. It will be easier to support these requirements using conventional technologies. Similarly, in blockchain systems, problems may arise such as disputed transactions, incorrect addresses, exposure or loss of private keys, data-entry errors, or unexpected changes to assets tokenized on blockchain. The immutability of blockchain ledgers may make them less adaptable than conventional technologies controlled by trusted third-party organizations that support rollback.

Using blockchain to achieve immutability and non-repudiation may be relatively expensive compared to other persistence mechanisms. There are existing mechanisms available to prove the originality of data, like hashing technology, and cryptographically signed data. In traditional database systems, the ACID properties (Atomicity, Consistency, Isolation, and Durability) are critical. However, for blockchains that use Nakamoto consensus (longest chain wins), the classic durability property does not hold because a transaction initially thought by a participant to be committed (i.e. on the longest chain) may later turn out to have been on a shorter chain, and so no longer be committed. Such blockchains only offer a long-run probabilistic durability property, and therefore are not immutable in a simple way. However, (a) switching to a longer chain is evident to participants, and (b) when a transaction has been committed to a blockchain for a sufficiently long time, it will in practice be immutable. Blockchains that use other consensus mechanisms (such as Practical Byzantine Fault Tolerance) can offer stronger, more conventional immutability properties. However, typically these consensus mechanisms can only be used where there is a small number of well-known nodes participating in the operation of the blockchain.

6.1.5 High Performance

Does the system need to support extremely short response times or process very large amounts of data? If so, conventional technologies may be more suitable than blockchain technology.

System performance usually relates to *latency* which is the system response time and *throughput* which is the aggregate system work rate. Blockchain systems such as Bitcoin and Ethereum cannot currently match the maximum throughput of conventional transaction processing systems such as the Visa payments network. This is a known and current limitation but is being addressed by the development of new mechanisms such as sharding, state channels, and reduced inter-block time. While blockchains are currently not highly scalable, this is not necessarily an

inherent limitation, and may be overcome in the future. Consortium and private blockchains with careful design and performance tuning have much better performance compared to public blockchains. When data has previously been written to the blockchain, read latency is the response time for accessing historical data from a blockchain client. Read latency can be much faster on blockchain than with conventional technologies, because clients can keep a full local copy of the database, and so there are no network delays. The request to write data into a blockchain is done by sending a transaction to the network. The write latency is probabilistic, and there are several sources of uncertainty. All blockchains will have small network delays. For blockchains with Nakamoto consensus, a node should not be highly confident that the most recent block it saw will ultimately be included in the main chain. So, to increase the confidence that data has successfully been committed to the blockchain, we can wait for a number of confirmation blocks. Waiting for more confirmation blocks will increase write latency.

Blockchains are inherently not suitable for storing Big Data, i.e. large volumes of data or high-velocity data. This is because on a blockchain there is massive redundancy in the large number of processing nodes holding a full copy of the distributed ledger. Big Data is hard to physically move in a distributed system, and the large numbers of replicas make it infeasible to store it on a blockchain.

6.1.6 Transparency

The third question in the design process is whether data transparency is required or acceptable in the system. *Data transparency* is the property that data is available and accessible to by other parties in the system. Examples include Facebook public newsfeed posts or Twitter public tweets. Anyone can access and read these posts. Social media such as Facebook or Twitter support *confidentiality* by allowing users to choose what they publish to the public or to specific audiences. Consider also the supply chain domain. Logistics efficiency can be improved by providing greater transparency on the status of shipments and processes, which are currently often opaque. Using blockchain in trade finance to evidence trade-related documents can reduce lending risk, and smart contracts can control inter-organizational process execution, and transparently automate delayed or instalment payments. However, very often customer relationships, pricing, or even aggregate transaction volume are commercially sensitive information that parties do not want to share widely.

Blockchain provides a neutral platform where all participants can see and audit the published data. This is important to guard integrity, with validation by all processing nodes. In a public blockchain, nodes validate that cryptocurrency transfers are from addresses that have enough cryptocurrency and signed with an authorized private key. For smart contracts, nodes validate that the effects of the smart contract program execution are correctly recorded on the blockchain. If data transparency is required or acceptable, a blockchain may be suitable. However, if data transparency is not acceptable, it can be difficult to use a blockchain to manage

that data. Confidentiality is harder to establish in blockchain-based systems, because information is visible to all participants.

Another confidentiality concern is the amount of interactions between parties. It is possible to create a new address for each transaction, but the flow of assets may still be used to infer relationships between addresses. Even if parties try to use pseudonyms, the contents of a transaction are publicly visible. Reuse of addresses and their connection via transfers of digital currency can provide opportunities to reidentify participants. Nonetheless, this limitation does not matter for all use cases. For example, public blockchains may be suitable as infrastructure for public advertising or fully open government registries, even in highly regulated industries. Consider that banks advertise on television, but television is not a highly regulated banking transaction system. Integrity in advertising may be required, but rather than privacy or confidentiality, publicity is important. Public blockchains can provide integrity and publicity. Other examples might include systems for secure software package management and IoT device configuration updates.

Sometimes, although raw data cannot be shared, it may be acceptable to share encrypted forms of that data, and in such cases a blockchain could be used. Information could be encrypted before being uploaded to the blockchain: asymmetrically with a particular party's public key, so that only this party can decrypt it, or symmetrically with a shared secret key, so that the group of parties with access to the secret key can decrypt it. The latter case requires a secure means of exchanging the secret key. Encrypting data before storing it on a blockchain may increase confidentiality, but will reduce performance and may harm independent auditability.

Encrypting data will make it difficult or impossible to use smart contracts with that data. If information needs to be processed by smart contracts, the information typically has to be decrypted. This is because smart contract code runs on all nodes of the network, and thus any of them needs to be able to process the input data. This is required to achieve consensus on the outcomes of smart contract execution. Embedding keys within a smart contract would reveal the keys to all participants of the blockchain network.

Sometimes encryption is not acceptable because there may be concerns about successful encryption key management or future technological developments in decryption (such as through quantum computing). Encrypted data may still reveal information as metadata, such as aggregate transaction volume.

Greater transparency is in tension with confidentiality, even if pseudonyms and encryption are used. Consortium and private blockchains can provide read access controls, but this will not provide commercial confidentiality between competitors on a consortium blockchain. The main trade-off is between the benefits of sharing data within the group of collaborators (visibility) and retaining confidentiality towards competitors where needed. In situations where full data transparency between all participants may not be acceptable, and where encrypting data is not acceptable or workable, a more-controlled data sharing can be enabled by distributed ledger technology platforms that are not full blockchains. Platforms such as R3's Corda or Hyperledger Fabric provide small ledgers shared between

parties of interest to each transaction. These platforms may be suitable where greater control is required over confidentiality.

6.2 Example Use Cases for Suitability Evaluation

This section uses the above evaluation framework to assess the suitability of using blockchain for four use cases. The first use case, supply chain, is aligned with the one described in Section 4.1. To illustrate other outcomes, we introduce three additional use cases in brief. Table 6.1 gives the summary of the evaluation results based on the seven questions. *Note that these results are illustrative only and should not be taken as valid guidance for real-world systems.*

6.2.1 Use Case 1: Supply Chain

A supply chain is the collection of processes involved in creating and distributing goods, from raw materials to completed products, through to consumers. According to a Deloitte survey, 42% of the companies in consumer goods and manufacturing planned to spend at least $5 million on blockchain technology in 2017.[1] Walmart has tested blockchain technology for their supply chain management in a pilot project that started on the first quarter of 2017 on tracking pork in the USA and China. The use of blockchain for supply chain is an extremely active area of innovation and technology development.

Supply chains are highly complex multiparty systems that span participants such as farmers, factories, transport providers, and retailers. Operations are distributed and often loosely coupled between participants. Data transparency is desired by participants to support logistics planning and to identify and respond to problems. Controlled confidentiality is required for open supply chain infrastructure, and this could be supported by the use of related-party ledgers in distributed ledger systems or by combining conventional information exchange technologies with hashed information on blockchains to ensure integrity and authorization. However, in vertically controlled supply chains, confidentiality can be managed by the use of a private blockchain. Transaction history and data immutability are desired to enable traceability back to the origin of goods and to control fraud and substitution. Current supply chain systems are often still paper-based, and thus cannot easily share information in real time. Digital solutions often only apply within vertically controlled parts of the supply chain, and information gaps can be created when subcontractors are used or when goods leave the scope of control. The time taken in

[1] https://www.bloomberg.com/news/articles/2016-11-18/wal-mart-tackles-food-safety-with-test-of-blockchain-technology.

Table 6.1 Results of the suitability evaluation of four example use cases

	Supply chain	Electronic health records	Identity	Stock market
Multiparty	Required	Required	Required	Required
Trusted authority	Not required	Decentralized	Not required	Not required
Centralized operation	Not required	Not required	Not required	Not required
Data immutability and non-repudiation	Required	Required	Required	Required
High performance	Not required	Not required	Not required	Required
Data transparency and confidentiality	Transparent (but not fully public)	Confidential	Transparent	Confidential
Sample result	DLT	Conventional system	Blockchain	Conventional system

Results are illustrative only, and should not be taken as ultimate guidance.

a supply chain is dominated by physical transportation and storage, which moderates demand for performance. Reasonably short latency is required at key points of handover of goods, but there is no requirement for extreme throughput or latency.

Supply chains are a promising area for blockchain-based applications. The complex, dynamic structure of business relationships and operations in a supply chain can be accommodated by the flexible structure of blockchain node networks, and the logically centralized view of information provided by a blockchain supports many of the demands for transparency in a supply chain.

6.2.2 Use Case 2: Electronic Health Records (EHRs)

Electronic health records (EHRs) are collections of patient medical records. They contain clinical data such as blood type, vital signs, past medical records, medications, and radiology reports for patients.[2] Currently, these records are often maintained by specific healthcare providers over time, in siloed systems not connected to other EHRs.

Multiple parties including patients, professionals, and organizations from different medical jurisdictions are involved in data exchange to allow more efficient healthcare and research. Healthcare service providers are decentralized trusted authorities. Each has access to patient data and has the authority to make the changes to that data. The operation of EHR systems is often distributed across healthcare service providers. Data transparency remains one of the main issues in existing EHRs. Patient privacy is critical, and normally information should only be shared with patient consent. Sometimes exceptions are made, for example, to access medical records in emergency situations, or to allow access to anonymized data for approved medical research. Accesses made to EHRs are often required to be logged for audit purposes. In addition to tight controls on read access, it is also important that health records cannot be inappropriately created or updated. EHRs do not typically need very low latency updates, and most patients' records do not change often. However, sometimes large diagnostic image information needs to be managed for an EHR.

Because of privacy constraints, blockchains are not normally used to store patient records directly, even in encrypted form. Instead, conventional systems are used to manage EHR source data, with blockchains providing auxiliary services. One example is the use of blockchains to keep audit logs of accesses made to EHRs. Records in these audit logs are typically encrypted or hashed to maintain patient privacy. MedRec[3] is an initiative to explore on blockchain architecture in contributing to secure and interoperable EHR systems. MedRec stores a pointer to

[2] https://www.cms.gov/Medicare/E-Health/EHealthRecords/index.html.

[3] https://medrec.media.mit.edu/.

patients' data in the blockchain and allows patients to choose when and with whom to share their data.

6.2.3 Use Case 3: Identity Management

Identity management underlies most business and social interactions. Individuals, organizations, devices, and assets can be identified by many schemes such as passports, wedding certificates, serial numbers, and registration certificates. An identity management system (IDM) manages user identities within an enterprise system. Conventionally, the operations of such systems are centralized and managed by a trusted authority. The authority sets permissions and roles for users to ensure they only access parts of the system relevant to them. Integrity is critical for IDM, to allow only authorized updates to users and their authorizations. Authorization can be complicated by requirements for delegated authorization and by requirements to enable dynamic revocation of authorizations. Logs of system accesses are often required, to be able to audit and investigate proper use of the system. Read accesses to an IDM can be frequent, to confirm authorized access, but updates to information in an IDM are normally much less frequent. It is often acceptable for there to be some delay in propagating updates to information about user identities and their authorizations.

Blockchain has been trialled for the management of individuals' identity for authorization, authentication, user role, and privileges within enterprise systems.[4,5] Blockchain allows the roles, permissions, and privileges of users to be verified by the distributed peers connected to the blockchain network. This removes the need for a centralized administrator and centralized database. Data on blockchain is transparent to everyone on the network by default. The immutable transaction history is duplicated to all connected peers. IDMs on a blockchain ensure that user identities, roles, and authorizations will not be altered improperly. Despite the fact that most current blockchains' performance does not match that of existing systems, it can still be viable to implement IDMs on blockchain because most operations require read access, which can have low latency for blockchains. Privacy is a critical requirement for IDMs, and so plaintext identity information for users is not normally stored directly on a blockchain. Instead, that is either kept off-chain or perhaps encrypted on-chain. For any solution, a significant privacy concern for system designers must be the possibility of reidentification attacks that may allow identities to be inferred from metadata or relationships stored on the blockchain.

[4] https://www.ibm.com/blogs/blockchain/2017/05/its-all-about-trust-blockchain-for-identity-management/.

[5] https://letstalkpayments.com/22-companies-leveraging-blockchain-for-identity-management-and-authentication/.

6.2.4 Use Case 4: Stock Market

A stock market is a place where stocks, bonds, and securities are traded. A stock market system inherently involves multiple entities to issue and trade stocks and conventionally is implemented by a centrally controlled and maintained register of stock ownership. In most jurisdictions, regulatory approval is required for the operation of stock market infrastructure, and regulatory approval may be required for the trading of specific stocks. In those contexts, the stock market is a natural trusted authority. Integrity, immutability, and non-repudiation are critical to ensure that high-value trades cannot be undone by either party. Transaction history is important in providing evidence for trades and current stock holdings. Stock markets typically have a high-volume, extremely low-latency price-setting mechanism to match buyers and sellers. However, stock markets typically settle trades (i.e. exchange the stocks and payment) at a later time. Settlement can have high throughput requirements but typically does not have extreme latency requirements.

Blockchain technology allows trades to be settled by the blockchain infrastructure using peer confirmation, removing the need for centralized operation and centralized authority to verify trades. Data transparency, however, is an issue for blockchains in the context of the stock market. All investors and market participants are exposed to blockchain participants. Even in a consortium blockchain between brokers, this creates a disadvantage to the investor and may be prohibited by a regulator. Transaction history is important because it keeps track of the ownerships of shares and also any changes that happen. Data immutability is also crucial as it ensures that no successful transactions can be tampered with by anyone. Looking at the scalability of existing stock exchanges, blockchain technology might not be suitable for this use case until the performance of blockchain can match up with current conventional technologies. Overall, blockchain is not highly suitable for the operation of conventional regulated stock markets. However, some blockchain solutions are being explored. NASDAQ offers its Linq blockchain ledger for registration and settlement of private securities,[6] and the Australian Stock Exchange (ASX) is also exploring distributed ledger technology to replace their current Clearing House Electronic Subregister System, for core modules such as trade registration and settlement.[7]

6.3 Design Process for Blockchain-Based Systems

In this section, we discuss an indicative model for the design of systems that might use blockchain technology. The process is shown in Fig. 6.2. Every step in the process is a procedure to decide between alternative options. The available

[6] http://ir.nasdaq.com/releasedetail.cfm?releaseid=948326.

[7] http://www.asx.com.au/services/chess-replacement.htm.

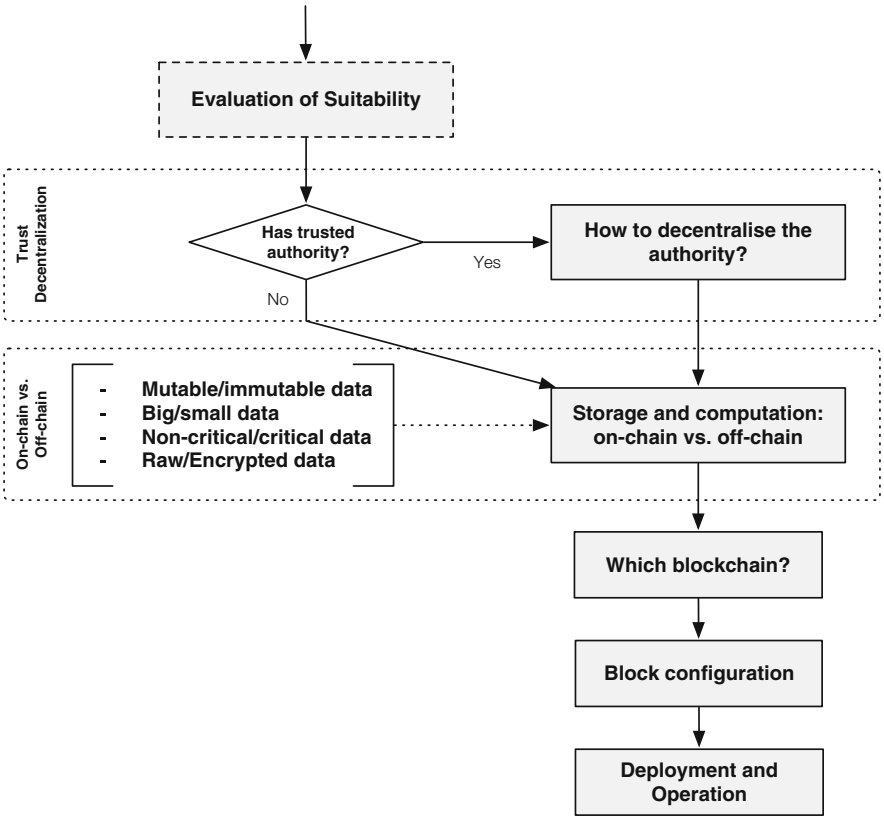


Fig. 6.2 Design process for blockchain-based systems. © 2017 IEEE. Reprinted, with permission, from Xu et al. (2017)

options discussed in Chapter 3 are used to assist decision-making and to guide the system design at different stages of the design process. This enables a systematic comparison of the capabilities of different design options. Chapter 3 describes the impact of the design options on quality attributes. Trade-off analysis between affected quality attributes is the foundation for the comparison of design options. The design process starts after the initial evaluation of blockchain suitability. The arrows illustrate one possible sequence of design decisions.

6.3.1 Trade-Off Analysis

As with any software system, there are trade-offs between quality attributes in the design of blockchain-based systems. Some decisions mainly affect scalability (like block size and frequency), security (like consensus protocol), cost efficiency (like type of blockchain), or performance (like data structure). Design decisions

that improve the performance of one quality attribute for a system may harm the performance of other quality attributes. Some simple examples of this include:

- Encrypting data before storing it on a blockchain may increase confidentiality, but will reduce performance, and may harm transparency or independent auditability.
- Storing only a hash of data on-chain and keeping the contents off-chain will improve confidentiality and may improve performance but partly undermines the distinctive benefit of blockchains in providing distributed trust. This may create a single point of failure, reducing system availability and reliability.
- Using a private blockchain instead of a public blockchain may allow greater control over the admittance of processing nodes and transactions into the system but will also increase barriers to entry for participation and thus partly reduce some of the benefit of using a blockchain.
- For blockchains that use Nakamoto consensus such as Bitcoin or Ethereum, waiting for a higher number of confirmation blocks may increase confidence in integrity and durability of transactions but will harm latency and thus may impact service availability.

6.3.2 Decentralization

According to the discussion in Section 6.1, a blockchain is used in scenarios where no single trusted authority is required or acceptable and where the trusted authorities can be decentralized or partially decentralized. For the deployment and operation of systems, there is a spectrum of options ranging from centralized monopolies to central parties with a competition between parties, to services provided jointly by a consortia, through to fully open service provision in a public peer-to-peer system. It is possible that some components or functions are decentralized while others are centralized. Design decisions regarding trust decentralization are discussed in Section 3.2.

6.3.3 On-Chain vs. Off-Chain

Blockchains are usually combined with other components in a broader system. Functionality such as user interfaces, cryptographic key management, IoT integration, and communication with other external systems is inherently off-chain. Many kinds of data are also better stored off-chain, for scalability reasons (big data), for confidentiality reasons (private data), or for dealing with legacy databases. Although we say ‘big data’ is not suitable for storing on a blockchain, even ‘not tiny’ data may be too large to feasibly store on a blockchain. Cost calculations can help to determine the resolution of design decisions for this issue (see also Chapter 9).

While blockchains provide some unique properties, the amount of computational power and data storage space available on a blockchain network remains limited. In addition, the monetary cost of using public blockchains follows a different cost model than conventional software systems. In regard to cost efficiency, performance, and flexibility, major design decisions in using a blockchain include choosing what data and computation should be placed on-chain and what should be kept off-chain. Table 6.2 captures some of these options, which are described in more detail below.

Data

A common practice for data management in blockchain-based systems is to store raw data off-chain and to store on-chain just metadata, small critical data, and hashes of the raw data. However, the applications of storing item data on blockchain are not just for integration with external data. There are various uses for wholly on-chain auxiliary data, including 'colored coins' which are a class of overlays on Bitcoin to represent and manage real-world assets.

A detailed discussion of on-chain data storage cost can be found in the respective cost chapter, in Section 9.1. Here we focus on a higher-level consideration as part of the design process.

In the Bitcoin blockchain, there are different ways to store data in transactions. This was not a core feature in the original design of Bitcoin but has now been incorporated with a specific command, called *OP_RETURN*. Table 6.2 compares this mechanism with alternatives. While it offers some level of flexibility, storing data on the Bitcoin blockchain is slow and costly and limited to 40 bytes.

Ethereum, on the other hand, theoretically allows storing arbitrary structured data of any size in a transaction directly. However, the size of a transaction is limited by the maximum size of a block, and in practice transactions typically need to be smaller to be accepted due to the transaction load from other users. In addition, Ethereum provides two other ways to store arbitrary data, using smart contracts. The first option is to store the data as a variable in a smart contract. The second option is to store arbitrary data as a log event of a smart contract. Storing data as a variable in a smart contract is more efficient to manipulate, but less flexible due to the constraints of the Solidity language on the value types and length. The flexibility and performance of using smart contract log events is intermediate because log events allow up to three parameters to be queried.

Finally, we reiterate that data storage on blockchain follows a different cost model than conventional data storage. Although it may seem more expensive, storing data on blockchain is a one-time cost for permanent storage. (However, note that Ethereum allows a partial refund on reclaimed smart contract variable storage.)

Selection of off-chain data storage concerns the interaction between the blockchain and the conventional data storage facilities. Off-chain data storage can be through conventional enterprise IT systems, a private cloud on the client's infrastructure, or a public storage provided by a third-party. The flexibility of using cloud to store data depends on the implementation. Some peer-to-peer data storage

Table 6.2 Design decisions regarding storage and computation with an indication of their relative impact on quality properties (⊕, Least favourable; ⊕⊕, Less favourable; ⊕⊕⊕, More favourable; ⊕⊕⊕⊕, Most favourable)

Design decision		Option	Impact			
			Fundamental properties	Cost efficiency	Performance	Flexibility
Data	On-chain	Embedded in transaction (Bitcoin)	⊕⊕⊕⊕	⊕	⊕	⊕⊕
		Embedded in transaction (Public Ethereum)		⊕⊕⊕⊕	⊕	⊕⊕⊕
		Smart contract variable (Public Ethereum)		⊕⊕	⊕⊕⊕	⊕
		Smart contract log event (Public Ethereum)		⊕⊕⊕	⊕⊕	⊕⊕
	Off-chain	Private/third-party cloud	⊕	~KB negligible	⊕⊕⊕⊕	⊕⊕⊕⊕
		Peer-to-peer system		⊕⊕⊕⊕	⊕⊕⊕	⊕⊕⊕
Computation	On-chain	Transaction constraints	⊕⊕⊕⊕	⊕	⊕	⊕
		Smart contract				
	Off-chain	Private/third-party cloud	⊕	⊕⊕⊕⊕	⊕⊕⊕⊕	⊕⊕⊕⊕

facilities are designed to be friendly to blockchain, such as IPFS[8] and Storj.[9] IPFS is free, but ensuring availability requires providing an IPFS server that hosts the data. The cost of Storj is US$0.015/GB/month. In a peer-to-peer data storage, the data is replicated automatically by the peer-to-peer network or based on the behaviour of users, e.g. data is replicated once a user accesses it. In a cloud environment, data replication needs to be managed by the system or consumer.

Computation

Computation in a blockchain-based system can be performed on-chain (e.g. through smart contracts) or off-chain. Different blockchains offer different levels of expressiveness for on-chain computation. For example, Bitcoin only allows simple scripts and conditions that must be satisfied to transfer Bitcoin payments. Ethereum allows more general (Turing complete) programs, and these programs can not only perform conditional payments but also make modifications to the working data in smart contract variables. There are other smart contract languages which are more expressive than Bitcoin's simple scripts, but which are purposefully not Turing complete, in order to facilitate static analysis. An example is the Digital Asset Modelling Language (DAML),[10] which is designed to codify financial rights and obligations.

Smart contracts are not processed until their invoking transactions are included in a new block. Blocks impose an order on transactions, thus resolving nondeterminism which might otherwise affect their execution results. One benefit of using on-chain computation, rather than using blockchain as a data layer only, is the inherent interoperability among the systems built on the same blockchain network. Other benefits are the neutrality of the execution environment and immutability of the program code once deployed. This facilitates building trust in the shared code among untrusting parties.

Other Considerations

Deciding between on-chain and off-chain not only depends on trade-offs among quality attributes, but also on how information and computation are used by other components in the broader system. Take identity information (Section 6.2.3) as an example. Identity supports systems where there is a requirement to know the individual human or system involved in transactions. Services such as international payments have regulatory requirements to establish the identity of participants, as part of Anti-Money Laundering (AML) and Counter-Terrorism Financing (CTF)

[8] https://ipfs.io/.

[9] https://storj.io/.

[10] https://digitalasset.com/press/introducing-daml.html.

policies. From a purely technical perspective, real-world identities are not necessarily required. For example on Bitcoin, transacting agents (which are not necessarily persons) are only cryptographically identified, pseudonymously. So international exchange of the Bitcoin digital currency can be performed without establishing real-world identity. Nonetheless, AML/CTF requirements are not obviated by the use of a blockchain. Identity is critical here, and identity on blockchain is sometimes considered to be a key enabler for many financial services on blockchain. However identity information does not necessarily need to be stored on-chain, off-chain protocols might be used instead. Privacy and confidentiality can be a challenge when integrating identity information into a blockchain-based system.

6.3.4 Blockchain Selection and Configuration

At this stage, a blockchain platform is selected according to the requirement of the use case and characteristics of blockchain platforms and trade-off analysis discussed in Chapter 3. Normally, the consensus protocol and some other decisions are fixed once a particular blockchain is selected. Hyperledger Fabric is an exception, where a modular architecture is used to support pluggable implementations of various consensus protocols. For some blockchain platforms, for example, those using a proof-of-work protocol, the inter-block time can be configured through adjustments to the difficulty of mining.

6.3.5 Deployment and Operation

Finally, the choice of where to deploy the modules of the blockchain-based system is also important for the quality attributes of blockchain-based systems. For example, deploying a blockchain on a cloud provided by a third-party, or using a blockchain-as-a-service model directly, introduces the uncertainty of cloud infrastructure into the system. Here the cloud provider becomes a trusted third-party and a potential single point of failure for the system. Deploying a public blockchain system on a virtual private network can make it a private blockchain, with permissioned access controls provided at the network level. However the virtual private network will introduce its own additional latency overhead.

There are specific design challenges related to the operation of blockchain-based systems, which architects should be aware of when deciding to use a blockchain. Blockchain-based systems can be harder to modify than conventional systems. The blockchain platform software runs on multiple independently operating nodes, and updating that software can be physically and administratively difficult to coordinate. The blockchain ledger is also immutable by design and so cannot be retrospectively updated to facilitate system modification. Similarly, in blockchain-based systems

that use smart contracts to regulate interactions between mutually untrusting parties, trust is derived partly from the fact that the code cannot be changed easily.

This inherently creates challenges for governance: the management of the evolution of blockchain-based systems. Changes may be made to correct defects, add features, or migrate to new IT contexts. However, in a multiparty system with no single owner, managing these changes is more like diplomacy than traditional risk management or conventional product management. Hence, the current configuration of blockchain is not suitable to implement on a system that may need to change or be modified frequently. Lessons may be drawn from governance in open-source software, which faces similar development challenges. However, the governance of a blockchain is not just a software development problem—it is also a deployment and operations problem. For both public and private blockchain systems, key stakeholders include the users of the blockchain, software developers with moral or contractual authority over the code base, miners or processing nodes in the blockchain ecosystem, and government regulators in related industries. However, blockchain immutability may also simplify governance oversight to some degree. For instance, smart contracts deployed on a blockchain will be resistant to tampering and will continue to be individually available for execution while the whole blockchain operates normally. These factors should be taken into consideration when deciding to use blockchain as a component.

6.4 Summary

Due to their fundamental properties and limitations, blockchains do not fit to all scenarios. Thus, before designing a system, the suitability of a blockchain needs to be evaluated against the system requirements. This chapter started with a suitability framework for assessing the suitability of using blockchain in a various contexts, based on the characteristics of the use case. After the suitability framework, a general process for designing blockchain-based applications was discussed. Throughout this design process, the available options discussed in Chapter 3 are used to assist decision-making and to guide the system design at different stages of the design process, by enabling a systematic comparison among the capabilities of different design options.

6.5 Further Reading

This chapter is partly based on our earlier works (Xu et al. 2017; Lo et al. 2017).

MedRec is an initiative to explore how a blockchain-based architecture can contribute to secure and interoperable EHR systems. More details of MedRec can be found in Azaria et al. (2016).

Chapter 7
Blockchain Patterns

with Cesare Pautasso and Qinghua Lu

In this chapter, we present a collection of patterns for the design of blockchain-based applications. In software engineering, a *design pattern* is a reusable solution to a problem that commonly occurs within a given context during software design. A design pattern defines constraints that restrict the roles of architectural elements (processing, connectors, and data) and the interaction among those elements. Adopting a design pattern causes trade-offs among quality attributes. Our pattern collection includes three patterns about interaction between blockchain and the external world, four data management patterns, three security patterns, and five contract structural patterns. The pattern collection provides architectural guidance for developers to build applications on blockchain. Figure 7.1 gives an overview of these patterns. Using the patterns in an application architecture can better align it with the unique properties provided by blockchain, avoid its limitations, and achieve other quality attributes.

The three patterns that describe different ways for blockchains to communicate data with the external world are *oracle* (Section 7.1.1), *reverse oracle* (Section 7.1.2), and *legal and smart contract pair* (Section 7.1.3). The patterns about managing data on and off blockchain are *encrypting on-chain data* (Section 7.2.1), *tokenization* (Section 7.2.2), *off-chain data storage* (Section 7.2.3), and *state channel* (Section 7.2.4). There are three patterns about the security of blockchain-based applications: *multiple authorization* (Section 7.3.1) and *off-chain secret enabled dynamic authorization* (Section 7.3.2) are aimed at adding dynamism to authorization of transactions and smart contracts, and *X-confirmation* (Section 7.3.3) further increases the security of transactions. The five structural patterns are concerned with the dependencies among and the behaviour of smart contracts. Smart contracts on blockchain are immutable. The challenge of how to upgrade a smart contract can hinder the evolution of blockchain-based applications.

X. Xu et al., *Architecture for Blockchain Applications*,
https://doi.org/10.1007/978-3-030-03035-3_7

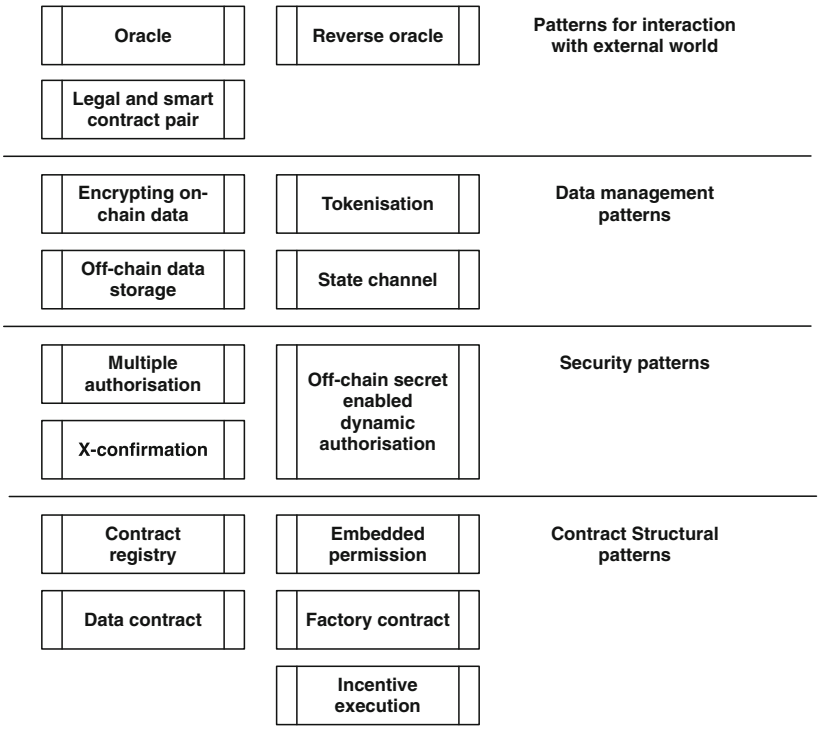


Fig. 7.1 Overview of the blockchain-based application pattern collection, adapted from Xu et al. (2018)

Contract registry (Section 7.4.1), *data contract* (Section 7.4.2), and *factory contract* (Section 7.4.4) are three patterns that target improved upgradability of smart contracts. *Embedded permission* (Section 7.4.3) aims to provide permission control of functions of smart contracts. Finally, *incentive execution* (Section 7.4.5) concerns maintenance of smart contracts.

In this chapter we follow an established form to describe each pattern, which includes the name of the pattern, a short summary, the context, the problem statement, an explicit discussion of the forces which make the problem difficult, the solution, its consequences, and some examples of known real-world uses of the pattern. Forces are identified with the corresponding quality attribute, as sometimes the solution will propose a trade-off between them. Regarding the consequences, we distinguish the benefits and drawbacks. Some of the discussions are only applicable to certain types or deployments of blockchain, such as monetary cost of data storage (public blockchains) and code execution (blockchains with smart contract capabilities).

7.1 Patterns on Interacting with the External World

Due to the unique properties and limitations of blockchain, a major architectural consideration for blockchain-based software applications is what data and executable code (smart contracts) should be kept on-chain and what should be kept off-chain. Two factors need particular attention, namely performance and privacy. Performance highly depends on the type of deployment of the blockchain. For example, a consortium blockchain can be configured to achieve much better performance than a public blockchain. As a component of a larger software system, blockchain needs to communicate data with other components within the software system (as shown in Fig. 5.1).

7.1.1 Pattern 1: Oracle

Summary Introducing the state of external systems into the closed blockchain execution environment.

Context From the software architecture perspective, a blockchain can be viewed as a component within a larger software system. In the case the blockchain is used as a distributed database for more general purposes other than purely blockchain-based services, the applications built on a blockchain will need to interact with other external systems. Thus, the validation of transactions on blockchain will depend on those external systems.

Problem The execution environment of a blockchain is self-contained. It can only access information present in the data and transactions on the blockchain. Smart contracts running on a blockchain are pure functions by design. The state of external systems is not directly accessible to smart contracts. Yet, function calls in smart contracts sometimes need to access state of the external world.

Forces

- *Closed environment.* Blockchain is a secure, self-contained environment, isolated from external systems. Smart contracts on blockchain cannot directly read the states of the external systems.
- *Connectivity.* In addition to the data found on the blockchain, general purpose applications might require information from external systems. For example, information such as geolocation information or weather data from a Web API[1] may be required.
- *Long-term availability.* While transactions on a blockchain are immutable, the external state used to validate a transaction may change or even disappear after the transactions were originally appended to the blockchain.

[1] https://openweathermap.org/api.

Solution To connect the closed execution environment of a blockchain with the external world, an *oracle* is introduced to evaluate conditions that cannot be expressed in a smart contract running within the blockchain environment. An oracle is a trusted third-party that provides smart contracts with information about the external world. When validation of a transaction depends on external state, the oracle is requested to check the external state and to provide the result to the validator (*miner*), which then takes the result provided by the oracle into account when validating the transaction. The oracle can be implemented inside a blockchain network as a smart contract with external state being injected into the oracle periodically by an off-chain *injector*. Other smart contracts can then access the data from the oracle smart contract. An oracle can also be implemented as a server outside the blockchain. Such an external oracle needs permission to sign transactions. To improve the reliability or trustworthiness of the oracle, a distributed oracle uses multiple servers. Figure 7.2 is a graphical representation of the pattern with the external oracle solution approach. Participants who wish to transact with each other on a blockchain could rely on an ad hoc arbitrator trusted by all the participants to resolve disputes or check external state. An arbitrator may be a human with a blockchain account who is able to sign transactions. Alternatively, an arbitrator may be automated and validate transactions based on state taken from the blockchain and the external world.

Consequences
Benefits:

- *Connectivity.* The closed execution environment of a blockchain is connected with the external world through the oracle. The applications based on blockchain can access external states through the oracle and use these external states in their execution.

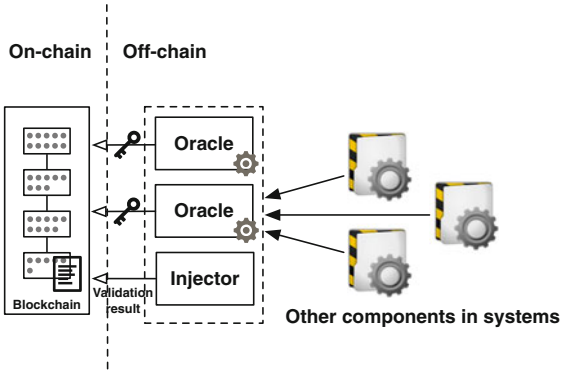


Fig. 7.2 Oracle pattern. This work is based on an earlier work: Xu et al. (2018) © ACM, 2018. http://dx.doi.org/10.1145/3282308.3282312. Included here by permission

Drawbacks:

- *Trust.* Using an oracle introduces a trusted third-party into the system. The oracle selected to verify or supply the external state needs to be trusted by all the participants involved in relevant transactions.
- *Validity.* External states injected into the transactions cannot be fully validated by other miners. Thus, when miners validate transactions including external state, they rely on the oracle.

Related Patterns *Reverse oracle* (Section 7.1.2)

Known Uses

- The concept of *oracle* is used in Bitcoin.[2] An oracle is a server outside the Bitcoin blockchain network which can evaluate user-defined expressions based on the external state.
- Orisi[3] is a distributed oracle scheme on Bitcoin. Orisi maintains a set of independent oracles and allows participants involved in a transaction to select a set of oracles and to define the quorum required before initiating a conditional transaction.
- Hyperledger Fabric *chaincode* (smart contracts) can in principle invoke any off-chain function, including to access external state. Chaincode is specified with *endorsement policies*, to specify which nodes are required to validate its execution. Chaincode with a singleton endorser node thus acts as a platform-supported oracle for Hyperledger Fabric. Endorsement policies can also specify M-of-N validation constraints, to act as platform-supported distributed oracles.
- Gnosis[4] is an example of arbitrator selection by participants. Gnosis is a decentralized prediction market that allows users to choose any oracle they trust, such as another user or a web service, e.g. for weather forecasts.

7.1.2 Pattern 2: Reverse Oracle

Summary The off-chain components of an existing system rely on smart contracts running on a blockchain to supply requested data and check required conditions.

Context In a software system, where a blockchain is one of the components, the off-chain components might need to use the data stored on blockchain and the smart contracts running on blockchain to supply data or check conditions.

Problem Many pre-existing or legacy systems do not have direct interfaces to blockchains. Data or functionality on a blockchain may have to be integrated with

[2] https://en.bitcoin.it/wiki/Contract#Example_4:_Using_external_state.

[3] http://orisi.org/.

[4] https://gnosis.pm/.

legacy systems, but a nonintrusive approach is required, not changing the core of the existing systems.

Forces

- *Connectivity.* Integrating blockchain into an existing system to leverage the unique properties or data of a blockchain.

Solution A component that can interact with both the blockchain and existing system components is added to the system. The reverse oracle component provides broader system functionality by mediating with blockchain data and smart contract functionality. Well-known smart contract functions can be configured in the component to access blockchain functionality, or the identity of transactions on the blockchain can be made visible to the system for integration. Figure 7.3 is a graphical representation of the pattern.

Consequences

Benefits:

- *Connectivity.* The blockchain is integrated into an existing system, either by configuring well-known smart contract functions to be invoked, or by making blockchain transactions visible to the system for integration.

Drawbacks:

- *Nonintrusive.* It is not always possible to use a blockchain in a nonintrusive way depending on the extensibility of the existing system. In particular, the probabilistic commit of blockchains using Nakamoto consensus may be inconsistent with normal transaction semantics in enterprise systems. Additional logic in the reverse oracle component may be required to cover these differences.

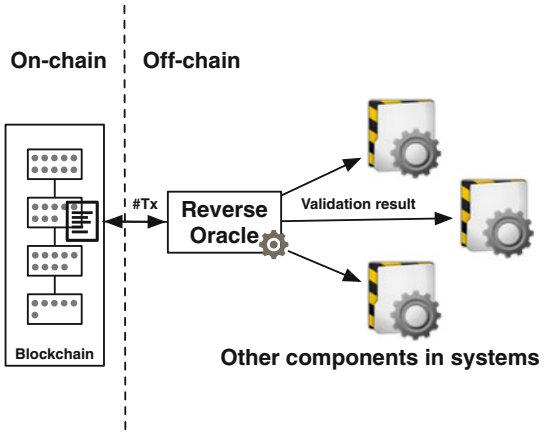


Fig. 7.3 Reverse oracle pattern. This work is based on an earlier work: Xu et al. (2018) © ACM, 2018. http://dx.doi.org/10.1145/3282308.3282312. Included here by permission

Related Patterns *Oracle* (Section 7.1.1)

Known Uses

- *Identitii*[5] provides a solution to enrich payments in banking systems with documents and attributes, using blockchain. Identitii uses the concept of identity token, which is an entity reference stored on a blockchain. Every payment is associated with an identity token, which is used to exchange enriched information about a payment. The identity token is exchanged between banks by being embedded into the SWIFT protocol.
- *Slock.it*[6] aims to build autonomous objects and a universal sharing network by using blockchain and IoT devices. Devices can sell or rent themselves and also pay for services provided by others. When renting a device, availability information is stored on blockchain; thus, validity checking is done using blockchain.

7.1.3 Pattern 3: Legal and Smart Contract Pair

Summary A bidirectional binding is established between a legal agreement and a corresponding smart contract.

Context The legal industry is becoming digitized, for example, using digital signatures has become a valid way to sign legal agreements. The Ricardian contract was developed in the mid-1990s as a concept for cryptographically identified legal contracts to also be machine interpretable. Digital legal agreements need to be executed and enforced.

Problem An independent trustworthy execution platform trusted by all the involved participants is needed to execute digital legal agreements. Blockchain can provide that platform, using on-chain smart contracts to digitize legal agreements.

Forces

- *Authoritative source.* A valid mapping is required between a legal contract and its corresponding smart contract, so that the smart contract can correspond with the authoritative legal contract.
- *Secure storage.* Blockchain provides a trustworthy data storage to keep the legal agreement.
- *Secure execution.* Blockchain provides a trustworthy computational platform that can execute digital agreements to enforce certain conditions as defined in a legal contract.

[5] https://identitii.com/.

[6] https://slock.it/.

Solution

A smart contract is created to implement some of the conditions defined in the legal agreement. When deployed, there is a variable to store the hash value of the legal agreement but initially has a blank value. The address of the smart contract is included in the legal agreement, and then the hash of the legal agreement is calculated and added to the contract variable. The immutability of the legal contract hash variable is implemented in custom code. By binding a physical agreement with a smart contract, the bridge between the off-chain physical agreement and the on-chain smart contract is established. The two-directional binding shows the intended mapping between the legal agreement and smart contract.

The smart contract digitizes some of the conditions defined in the agreement. These conditions can be checked and enforced automatically by the smart contract. However, not all legal terms can be digitized. The smart contract can also facilitate automated regulatory compliance checking, but extent of this might be limited depending on the data represented on the blockchain and on constraints of smart contract programming language. Figure 7.4 is a graphical representation of the pattern.

Consequences

Benefits:

- *Automation.* Some of the conditions defined in the legal contract, for example, a conditional payment, can be automatically executed or enforced by blockchain.
- *Audit trail.* Blockchain permanently records all historical transactions related to the legal contract and the smart contract. This immutable data enables auditing of the contract and its execution.

Drawbacks:

- *Expressiveness.* Smart contracts are written in programming languages. These programming languages might not be able to express all contractual terms or regulatory compliance conditions.

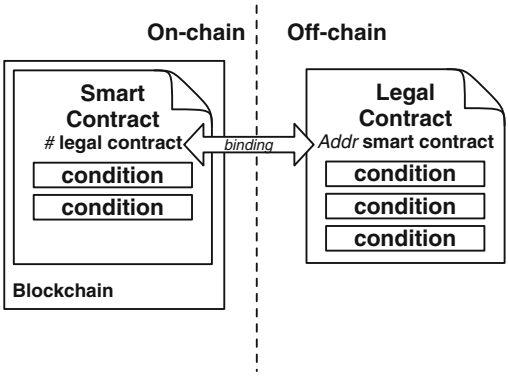


Fig. 7.4 Legal and smart contract pair pattern. This work is based on an earlier work: Xu et al. (2018) © ACM, 2018. http://dx.doi.org/10.1145/3282308.3282312. Included here by permission

- *Enforceability.* If a public blockchain is used, there is no central administering authority to decide on disputes nor to enforce court judgements.
- *Interpretation.* There might be many possible ways to interpret contract conditions and encode them in smart contracts. Ambiguity in natural language makes it a challenge to accurately implement legal terms in a way that will be agreed upon by all the involved participants.

Related Patterns N/A

Known Uses

- Ricardian contracts were not defined using blockchain smart contracts but have subsequently inspired approaches to using blockchain-based smart contracts for legal contracts in systems such as Corda[7] and EOS.[8] The *Smart Contract Template* proposed by Barclays[9] uses legal document templates to facilitate smart contracts running on Corda blockchain platform.
- Specific proposals for the representation of machine-interpretable legal terms have been explored in KWM's project on digital and analogue (*DnA*) contracts[10] and in the *Accord Project*.[11] Academic work has proposed logic-based languages to declaratively define smart contracts on blockchain.
- *Open Law*[12] is a platform that allows lawyers to make legally binding and self-executable agreements on the Ethereum blockchain. The legal agreement templates are stored on a decentralized data storage, IPFS.[13] Users can create customized contracts for specific uses.

7.2 Data Management Patterns

This section discusses three data management patterns that manage data on and off blockchain.

7.2.1 Pattern 4: Encrypting On-Chain Data

Summary Ensure confidentiality of the data stored on blockchain by encrypting it.

[7] https://www.corda.net/.
[8] https://eos.io/.
[9] https://www.barclays.co.uk/.
[10] https://github.com/KingandWoodMallesonsAU/Project-DnA.
[11] https://www.accordproject.org/.
[12] http://openlaw.io/.
[13] https://ipfs.io/.

Context For some blockchain applications, commercially sensitive data should be only accessible to specific participants. An example would be a special discount price offered by a service provider to a subset of its users. Such information might not be supposed to be accessible to the other users who do not get the discount.

Problem Data privacy is one of the main limitations of blockchain. All the information on blockchain is publicly available to participants. There is normally no privileged user within a blockchain network. On a private or consortium blockchain, the ability of parties to participate might be limited by the consortium agreement and by network access controls, but all participants will normally be able to see the full blockchain history. On a public blockchain, new participants can join the blockchain network freely.

Forces

- *Transparency.* Every participant within a blockchain network is able to access all the historical transactions on blockchain. This enables them to all collectively validate previous transactions. The transactions on a public blockchain are accessible to everyone, using blockchain explorer tools such as Etherscan.[14]
- *Lack of confidentiality.* Since all the information on blockchain is publicly available to everyone in the network, commercially sensitive data meant to be kept confidential should not be stored on blockchain in plain form.

Solution Symmetric or asymmetric encryption can be used to encrypt data before inserting the data into blockchain. One possible design for sharing encrypted data among multiple participants is as follows. First, one of the participants creates a secret key for encrypting data and distributes it during an initial key exchange. When one of the participants needs to add a new data item to the blockchain, they first symmetrically encrypt it using the secret key. Only the participants allowed to access the transaction are given the secret key and can decrypt the information. Figure 7.5 is a graphical representation of the pattern.

Consequences
Benefits:

- *Confidentiality.* Using encryption, the publicly accessible information on a blockchain is encrypted, so that is not readable by anyone who does not hold the secret key.

Drawbacks:

- *Key management.* Both symmetric and asymmetric encryption require off-chain key management. If key management is not done properly, it can lead to loss or disclosure of private or secret keys. If the required private key or secret key is compromised, the encryption mechanism will not protect the sensitive information.

[14] http://etherscan.io.

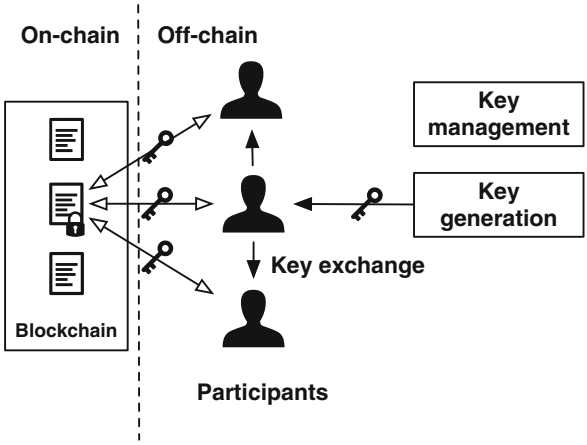


Fig. 7.5 Encrypting on-chain data pattern. This work is based on an earlier work: Xu et al. (2018) © ACM, 2018. http://dx.doi.org/10.1145/3282308.3282312. Included here by permission

- *Access revocation.* Revoking read access is a challenge after the encrypted data has been published to the blockchain. It is difficult to ensure that a party has destroyed their knowledge of a secret key. The encrypted data on a blockchain is immutable, and so as long as the participant retains the secret key, it retains access to the encrypted data.
- *Immutable data.* Even if stored in encrypted form, the sensitive data will remain in the blockchain forever. In addition to the risk of key compromise, the encrypted data may be subject to brute force decryption attacks at some time in the future. Breakthroughs in technology like quantum computing might render current encryption technologies ineffective. So even if the data is considered to be secure with a given key size when it is stored in the blockchain, this may no longer be the case in the future.
- *Key sharing.* The encryption key needs to be shared before the encrypted data on the blockchain can be read. Although blockchain itself can be used as a software connector to communicate data, secret keys cannot be shared in plain form through blockchain because the shared key would be publicly accessible if being communicated through blockchain.

Related Patterns N/A

Known Uses

- *Oraclize*[15] is a smart contract running on Ethereum public blockchain, which provides a service to access state from the external world. Oraclize allows smart contract developers to encrypt the parameters of their queries locally by using a

[15] https://blog.oraclize.it/encrypted-queries-private-data-on-a-public-blockchain-71d893fac2bf.

public key before passing them to a smart contract. The only one who can decrypt the call parameters is Oraclize, using the paired private key.
- *Crypto digital signature* has been suggested by *MLG Blockchain*[16] to encrypt data and share the data between the parties who interact through blockchain.

7.2.2 Pattern 5: Tokenization

Summary Using tokens on blockchain to represent transferable digital or physical assets or services.

Context Physical tokens such as tickets, share certificates, and casino chips are commonplace examples of representations of assets or services. These tokens (as paper documents or plastic chips) can be physically transferred between parties. Holding or redeeming a token will allow access to the assets or services represented by the token. The underlying assets can be digital or physical. Digital tokens can be electronically communicated between parties, but like physical tokens can represent digital or physical assets or services.

Problem Tokens representing assets or services should be transferable, so that they are no longer held by the original party after the transfer. The holding of a token by someone should be able to be authoritatively determined by others.

Forces

- *Representation.* Rather than holding an underlying asset, which might be risky or physically difficult, a token represents the asset and is easy to handle.
- *Holding and transfer.* For tokens to function as property or to support other exclusive rights, it must be possible to determine whether someone holds a token, and it must be possible to transfer the token, so that the original party no longer holds the token.

Solution Blockchain provides a trustworthy platform to realize tokenization. There are different ways to implement tokenization using blockchain. Native tokens exist on public blockchains (e.g. BTC on Bitcoin, ETH on Ethereum), but in addition to being cryptocurrency, they can also represent other assets or services using transaction identifiers or other auxiliary data. Cryptocurrency transfers on the blockchain are then also interpreted as a transfer of those assets or services. However, using blockchain cryptocurrency as tokens is limited because there is little expressive power to represent assets, and there can be limitations on checking token transfer conditions.

A more flexible solution is to define tokens as a data structure in a smart contract. For an asset, tokenization is a process starting from an asset (e.g. money) being held in custody (e.g. at a bank) and being represented as this data in the smart

[16] https://mlgblockchain.com/crypto-signature.html.

contract. The smart contract imposes constraints to ensure that holding and transfer of the token support the requirements of the token scheme. Depending on the token scheme, transfer of the token may correspond to transfer of ownership of the asset, or perhaps some other right to use the asset. Conditions programmed into the smart contract can enforce conditions on the transfer of tokens.

Consequences
Benefits:

- *Representation.* Tokens implemented on blockchain, especially when using smart contracts, have a great range of expressive power suitable for representing many kinds of assets or services.
- *Holding and transfer.* Blockchain transactions record the transfer of tokens and ensure that a token cannot be 'double spent'. The transparency of a blockchain allows all participants to inspect the latest state of token holdings.

Drawbacks:

- *Integrity.* Integrity of tokens is guaranteed by the blockchain infrastructure, but bugs in smart contracts can lead to problems in the holding or transfer of assets. Even if the digital token is secure, the authenticity of the corresponding physical or digital asset is not guaranteed automatically.
- *Legal processes for ownership.* A token on a blockchain is not necessarily the authoritative source of information about the ownership of a physical asset. The owner of an asset may be entitled to sell the asset without being required to create a transaction on the blockchain. Also, legal processes such as court orders and bankruptcy proceedings can change the ownership of physical assets without any associated transaction being recorded on the blockchain.

Related Patterns *Reverse oracle* (Section 7.1.2)

Known Uses

- *ColoredCoin*[17] is an open source protocol for tokenizing digital assets on Bitcoin blockchain.
- *Ethereum token standards.* Twenty-four percent of the existing financial smart contracts on Ethereum use tokenization. The ERC20[18] token standard has been proposed for fungible tokens and describes the functions and events that a token smart contract should implement. Other standard interfaces have been defined for other token types, such as unique or serialized assets.
- *Digix*[19] uses tokens to track the ownership of gold as a physical asset.

[17] http://coloredcoins.org/.
[18] https://theethereum.wiki/w/index.php/ERC20_Token_Standard.
[19] https://digix.global/.

7.2.3 Pattern 6: Off-Chain Data Storage

Summary Use hashing to ensure the integrity of arbitrarily large datasets which may not fit directly on the blockchain.

Context Some applications consider using the blockchain to guarantee the integrity of large amounts of data.

Problem The blockchain, due to its full replication across all participants of the blockchain network, has limited storage capacity. Storing large amounts of data within a transaction may be impossible due to the limited size of the blocks of the blockchain. For example, Ethereum has a block gas limit to constrain the number, computational complexity, and data size of the transactions included in any block. Data cannot take advantage of the immutability or integrity guarantees without being stored on the blockchain.

Forces

- *Scalability.* Blockchain provides limited scalability because every bit of data is replicated across all nodes, where it is kept permanently.
- *Cost.* If a public blockchain is used, storing data on blockchain costs money (cryptocurrency), although the cost is a one-time cost to write the data. This is in contrast to traditional distributed data storage, like cloud, where costs are based on the amount of allocated storage space over time. A piece of data can be stored on blockchain by being embedded in a transaction, as a variable in a smart contract or as a log event. Storing data in a contract is an effective way to enable its manipulation but can have constraints from the smart contract languages on the value types and length. Different blockchains have different cost models for storing data.
- *Size.* There are limits on transaction size or block size. For example, on the Bitcoin blockchain, the default client only relayed *OP_RETURN* transactions up to 80 bytes, which was reduced to 40 bytes in February 2014.[20] Ethereum has a block gas limit that limits the sum of gas all transaction in a block are allowed to use.

Solution A blockchain can be used as a general purpose replicated database, as transactions logged in the blockchain can include arbitrary data on some blockchain platforms. For data of big size (essentially data that is bigger than its hash value), rather than storing the raw data directly on blockchain, a representation of the data with smaller size can be stored on blockchain with other small-sized metadata about the data (e.g. a URI pointing to it). The solution is to store a hash value (also called digest) of the raw data on chain. The value is generated by a hash function, e.g. one from the SHA-2 family, which maps data of arbitrary size to data of fixed size. Hash functions are one-way functions which are easy to compute, but hard to invert. If

[20] https://github.com/bitcoin/bitcoin/pull/3737.

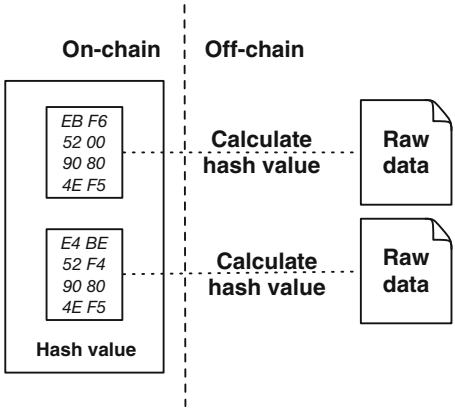


Fig. 7.6 Off-chain data storage pattern. This work is based on an earlier work: Xu et al. (2018) © ACM, 2018. http://dx.doi.org/10.1145/3282308.3282312. Included here by permission

even one bit of the input data changes, its corresponding hash value would change radically. A hash value can be used as a check to ensure the integrity of the raw data stored off-chain. The hash value recorded immutably in a blockchain transaction guarantees the integrity of the hash value as well as the original raw data from which the hash was derived. Figure 7.6 is a graphical representation of the pattern solution. Depending on the context, the hash value might double as (part of) a URI.

Consequences
Benefits:

- *Integrity.* Blockchain guarantees the integrity of the hash value that represents the raw data. The integrity of the raw data can be checked using the on-chain hash value.
- *Cost.* If a public blockchain is used, blockchain is utilized at a lower cost (fixed cost as the size of the hash value is fixed) for integrity of data with arbitrary size.

Drawbacks:

- *Integrity.* The raw data is stored off-chain, where the off-chain data store might not be as secure as blockchain. The raw data might be changed without authorization. This change will be detected, thanks to the hash of the original data stored on the blockchain. However, without additional measures, it will neither be possible to recover the original data nor to prevent the change from happening in the first place.
- *Data loss.* Since the raw data is stored off-chain, it may be deleted or lost. Only its hash value remains permanently on the blockchain.
- *Data sharing.* The on-chain data can be shared through using blockchain platforms. Additional communication mechanisms and storage platforms are required for data sharing off-chain.

Related Patterns N/A

Known Uses

- *Proof-of-Existence (POEX.IO*[21]*).* This service allows entering an SHA-256 cryptographic hash of a document into the Bitcoin blockchain as a 'proof-of-existence' of the document at a certain time. The hash value guarantees the data integrity of the document.
- *Chainy*[22] is a smart contract running on Ethereum blockchain. Chainy stores a short link to an off-chain file and its corresponding hash value.

7.2.4 Pattern 7: State Channel

Summary Transactions that are too small in value relative to a blockchain transaction fee or that require much shorter latency than can be provided by a blockchain are performed off-chain with periodic recording of net transaction settlements on-chain. Micropayments are a typical example of such transactions, but many other kinds of state updates or off-chain protocols can be treated in a similar way.

Context Micropayments are payments that can be as small as a few cents and very frequently executed. For example, payment of a very small amount of money to a Wi-Fi hotspot might be made frequently for small amounts of Wi-Fi data usage. Blockchain can back these kinds of transactions, but it is not necessary and cost-effective to store all such transactions on the blockchain.

Problem The decentralized design of blockchain has limited performance. Transactions can take several minutes or even 1 h (for Bitcoin blockchain) to be *committed* on the blockchain. Due to the long commit time and high transaction fees on a public blockchain (where fees are largely independent of the transacted amount), it is often infeasible to store many low-value transactions on the blockchain network. During a recent peak in demand, the average fee per transaction rose to the equivalent of US$55[23] on Bitcoin. On-chain transactions are suitable for transactions with medium to large monetary value, relative to the transaction fee.

Forces

- *Latency.* Blockchain transactions may take a long time to be committed, while users expect many kinds of transactions to happen instantaneously.
- *Throughput.* Blockchain has limited throughput scalability because every bit of data is replicated across all nodes and kept permanently.

[21] https://poex.io/.

[22] https://chainy.info/.

[23] Recorded for 22 Dec 2017 by https://bitinfocharts.com/comparison/bitcoin-transactionfees.html.

- *Cost.* Storing data on a public blockchain costs money (cryptocurrency). The transaction fee of an individual micropayment transaction might be higher than the monetary value associated with the micropayment transaction.

Solution Storing every low-value transaction on blockchain is infeasible due to the high relative cost of transaction fees. The state channel solution is to establish an agreed off-chain protocol between two participants, with a deposit from one or both locked up as security in a smart contract for the lifetime of the channel. The state channel keeps the intermediate states of the small transactions off-chain, and only stores the finalized aggregated (net) transaction on chain. The frequency of transaction settlement depends on the use case and agreement between the two sides. For example, in scenarios around utilities, internet service providers or electricity companies might establish payment channels with their consumers for an agreed monthly billing period. As the consumer uses data or energy daily, the intermediate state is stored in the off-chain state channel until the end of the month, when the channel is closed to finalize the payment for the whole month. A network of micropayment channels can be built where the transactions transferring small values occur off-chain. The individual transactions take place entirely off the blockchain and exclusively between the participants, across multiple hops where needed. Only the final transaction that settles the payment for a given channel or set of channels is submitted to the blockchain. The technologies used to implement state channels are specific to each blockchain platform. For example, the Lightning network[24] on the Bitcoin blockchain is a proposed implementation of Hashed TimeLock Contracts (HTLCs)[25] with bidirectional payment channels which allows secure payments across multiple peer-to-peer channels. A HTLC is a type of payments that use the features of Script, like *hashlocks* and *timelocks*, to require that the receiver of a payment acknowledges receiving the payment prior to a deadline by generating cryptographic proof. Figure 7.7 is a graphical representation of the pattern. Such off-chain channels could be generalized to exchange state for more general purposes other than monetary value.

Consequences
Benefits:

- *Speed.* Without involving the blockchain for every transfer, off-chain transactions can be settled without waiting for the blockchain network to process and commit each transaction.
- *Throughput.* The number of off-chain transactions that can be processed is not limited by the configuration of blockchain, such as the block size, block interval, or gas limit, and thus a much higher total throughput can be achieved than for on-chain transactions.

[24] https://lightning.network/.

[25] https://en.bitcoin.it/wiki/Hashed_Timelock_Contracts.

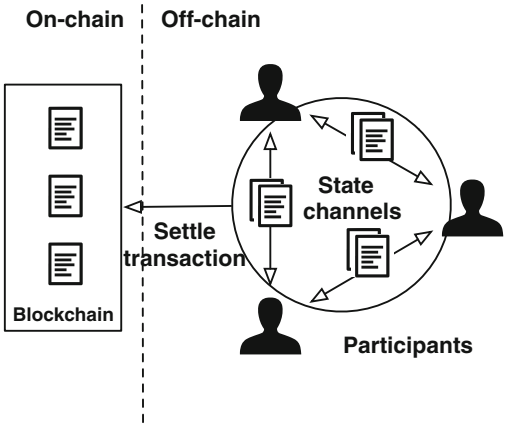


Fig. 7.7 State channel pattern. This work is based on an earlier work: Xu et al. (2018) © ACM, 2018. http://dx.doi.org/10.1145/3282308.3282312. Included here by permission

- *Privacy.* Other than the final settlement transaction, the individual off-chain transactions do not show up in the public ledger; thus, the detail of these intermediate off-chain transactions is not publicly visible.
- *Cost.* If a public blockchain is used, only the initial and the final transaction incur a transaction fee to be included in the blockchain. Individual off-chain transactions do not have blockchain transaction fees. Multi-hop off-chain transactions may be charged small transaction fees to compensate for reduced liquidity of channel providers, which are typically charged as a percentage of the transacted amount.

Drawbacks:

- *Trustworthiness.* Individual off-chain transactions might not be as trustworthy as the on-chain transactions because the transactions are not stored in the blockchain's immutable data store. The intermediate states of a state channel might be lost after the channel is closed.
- *Reduced liquidity.* To establish a payment channel, money from one or both sides of the channel needs to be locked up in a smart contract for the lifetime of the payment channel. The liquidity of the channel participants is thereby reduced.
- *Wallet.* A new wallet or extensions to existing wallets may be needed to support off-chain protocols.

Related Patterns N/A

Known Uses

- The *Lightning network* uses an off-chain protocol to enable micropayments of Bitcoin and several other cryptocurrencies. Micropayments are enabled by establishing a bidirectional payment channel through committing a funding transaction to the blockchain. This can be followed by a number of micropayment transactions that update the distribution of the funds within the channel without broadcasting transactions to the blockchain network. The payment channel can be closed by broadcasting the final version of the funding transaction to settle the payment.

- The *Raiden Network*[26] on the Ethereum blockchain is somewhat similar to the Lightning network. The basic idea is to avoid the consensus bottleneck by leveraging a network of off-chain payment channels that allow to securely transfer monetary value. Smart contracts are used to deposit value into the payment channels.
- *Orinoco*[27] is a payment channel solution built on the Ethereum blockchain. Other than payment channels, Orinoco also provides a payment hub for payment channel management. However, the payment hub introduces an extra party that needs to be trusted by both the sender and the recipient of the payment channel.
- *State channel* on Ethereum[28] and *Gnosis Go*[29] offer a more generalized form of state channels that support exchanging state for general purpose applications.

7.3 Security Patterns

This section discusses three security patterns that mainly concern the security of blockchain-based applications.

7.3.1 Pattern 8: Multiple Authorization

Summary A set of blockchain addresses which can authorize a transaction is predefined. Only a subset of the addresses is required to authorize transactions.

Context In blockchain-based applications, activities might need to be authorized by multiple blockchain addresses. For example, a monetary transaction may require authorization from multiple blockchain addresses.

Problem

- The actual addresses that authorize an activity might not be able to be decided in advance, due to sporadic or limited availability of some authorities.

Forces

- *Flexibility.* The actual authorities who authorize the transaction can be from a set of predefined authorities.
- *Tolerance of compromised or lost private key.* Authentication on blockchain uses digital signature. However, blockchain does not offer any mechanism to recover a lost or a compromised private key. Losing a key results in permanent loss of control over an account, and potentially smart contracts that refer to it.

[26] https://raiden.network/.

[27] http://www.orinocopay.com/.

[28] http://www.jeffcoleman.ca/state-channels/.

[29] https://forum.gnosis.pm/t/how-offchain-trading-will-work/63.

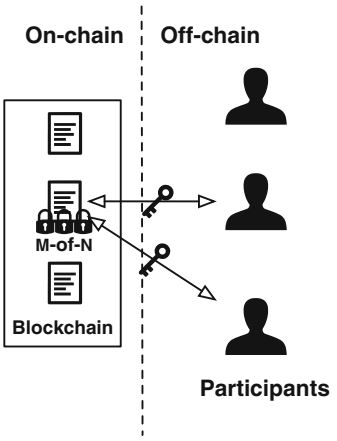


Fig. 7.8 Multiple authorization pattern. This work is based on an earlier work: Xu et al. (2018) © ACM, 2018. http://dx.doi.org/10.1145/3282308.3282312. Included here by permission

Solution On the Bitcoin blockchain, a multi-signature mechanism can be used to require more than one private key to authorize a Bitcoin transaction. In Ethereum, smart contracts can mimic multi-signature mechanisms. More flexibly, an M-of-N multi-signature can be used to define that M out of N private keys are required to authorize the transaction. M is the threshold of authorization. This on-chain mechanism enables more flexible binding of authorities. Figure 7.8 is a graphical representation of the pattern.

Consequences
Benefits:

- *Flexibility.* This pattern enables flexible binding of authorities but depends on the availability of authorities when the activity is conducted.
- *Lost key tolerant.* One participant can own more than one blockchain address to reduce the risk of losing control over their smart contracts due to a lost private key. In a smart contract implementation, of this pattern, there could be a function to update the list of allowed authorities and the authorization quorum. This update function may also require a quorum.

Drawbacks:

- *Predefined authorities.* Although the pattern enables flexible binding, all the possible authorities still need to be known in advance of any decision or update.
- *Lost key.* At least M private keys among the N private keys should be kept safely to avoid losing control.
- *Cost of dynamism.* If a public blockchain is used, updating the list of authorities costs money (cryptocurrency), as does deploying the logic for multiple authorities. There is greater cost for storing multiple addresses compared to only one.

Related Patterns *Off-chain secret enabled dynamic authorization* (Section 7.3.2). An off-chain secret enabled dynamic authorization pattern is used when the possible authorities are unknown beforehand.

Known Uses

- Multisignature mechanism provided by Bitcoin.[30]
- Multisignature wallet, written in Solidity and running on the Ethereum blockchain, is available in the Ethereum dapp browser Mist.[31]

7.3.2 *Pattern 9: Off-Chain Secret Enabled Dynamic Authorization*

Summary Using a hash created off-chain to dynamically bind authority for a transaction.

Context In blockchain-based applications, some activities need to be authorized by one or more participants that are unknown when a first transaction is submitted to blockchain.

Problem Sometimes, the authority who can authorize a given activity is unknown when the corresponding smart contract is deployed or the corresponding transaction is submitted to the blockchain. Blockchain uses digital signatures for authentication and transaction authorization. Blockchain does not support dynamic binding with an address of a participant which is not initially defined in the respective transaction or smart contract. All accounts that can authorize a second transaction have to be defined in the first transaction before that transaction is added to the blockchain.

Forces

- *Dynamism.* Dynamically binding one or more unknown authorities with a second transaction representing an activity after the first transaction was submitted to blockchain.
- *Predefined authorities.* Using only on-chain mechanisms, all the possible authorities are required to be defined beforehand.

Solution An off-chain secret can be used to enable a dynamic authorization when the participant authorizing a transaction is unknown beforehand. In the context of payment, for example, a smart contract can be used for escrow. When the sender deposits the money to the escrow smart contract, the hash of a secret (e.g. a random string, called *pre-image*) is also submitted with the money. Whoever receives the secret off-chain can claim the money from the escrow smart contract by revealing the secret. With this solution, the receiver of the money does not need to be defined

[30] https://en.bitcoin.it/wiki/Multisignature.

[31] https://github.com/ethereum/mist.

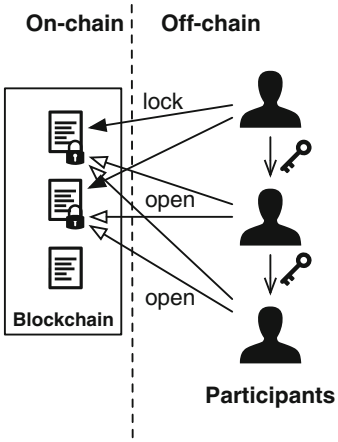


Fig. 7.9 Off-chain secret enabled dynamic authorization pattern. This work is based on an earlier work: Xu et al. (2018) © ACM, 2018. http://dx.doi.org/10.1145/3282308.3282312. Included here by permission

beforehand in the escrow contract. This can be generalized to any transaction that needs authorization from a dynamically bound participant. Note that once the secret is revealed, it cannot be reused. One variant is to lock multiple transactions with the same secret—by unlocking one, all of them are unlocked. Figure 7.9 is a graphical representation of the pattern. This solution is also referred to as *Hashlock*.

Consequences
Benefits:

- *Dynamism.* This pattern enables dynamic binding of unknown authorities after the transaction is added into the blockchain.
- *Lost key tolerant.* No specific private key is required to authorize transactions.
- *Routability.* This pattern has the useful property that once the secret is revealed, any other transactions secured using the same secret can also be opened. This makes it possible to create multiple transactions that are all locked by the same secret. This property is used by micropayment channels to enable multi-hop transfers where the money hosted by every hop and secured by a same secret can be released after the end receiver claims the money with the secret (i.e. the secret is revealed). The secret can be exchanged through off-chain channels.
- *Interoperability.* There is no need for a special protocol to exchange the secret. The secret can be exchanged in any way off-chain. This provides a mechanism for other systems to trigger events on blockchain.

Drawbacks:

- *One-off secret.* The secret used in this pattern is a one-off secret. Verification of the secret is on-chain. Thus, once a secret is embedded in a transaction submitted to the blockchain, the secret is revealed.
- *Combination of signature and secret.* Because this pattern has the property that once the secret is revealed, any other transactions secured using the same

secret can also be opened, sometimes the transaction protected by the secret should also be associated with a public key so that both a correct secret and an appropriate signature with the respective private key are required to authorize the transaction. This is applicable to the situation where a large set of authorities is known beforehand, but not all of them are allowed to authorize a certain activity/transaction. Thus, a hash secret is used to dynamically bind one or multiple authorities from the larger predefined set of authorities.

- *Lost secret.* The sender/initiator of a transaction takes the risk of losing the off-chain secret. If the secret is lost, the transaction cannot be authorized and being proceeded anymore. In the case of money transfer, the money associated with the transaction would be locked forever if the transaction cannot be authorized properly.
- *Man-in-the-middle attack.* A man-in-the-middle attack is possible when the transaction that reveals the secret is in the transaction pool of a miner (not included in the blockchain yet).

Related Patterns *Multiple authorization* (Section 7.3.1). The multiple authorization pattern is used when all the possible authorities are known beforehand. Multiple authorization pattern is an on-chain mechanism.

Known Uses

- *Raiden Network*[32] is a network of off-chain payment channels on top of Ethereum blockchain network, which enables secure value transfer. The multi-hop transfer mechanism in Raiden Network uses *hashlocked* transactions to securely route payments through a middleman.
- In the Bitcoin ecosystem, *atomic cross-chain trading*[33] allows one cryptocurrency (e.g. Bitcoin) to be traded for another cryptocurrency (e.g. tokens on a Bitcoin sidechain) using an off-chain hash secret.

7.3.3 Pattern 10: X-Confirmation

Summary Waiting for sufficiently many blocks as confirmations to ensure that a transaction added into blockchain is immutable with high probability.

Context The immutability of a blockchain using Nakamoto consensus is only probabilistic immutability. There is always a chance that the most recent few blocks are replaced by a competing chain fork.

Problem At the time a fork occurs, there is usually no certainty as to which branch will be permanently kept in the blockchain and which branches will be discarded. The transactions that were only included in the unsuccessful branches turn out not to

[32] https://raiden.network/.

[33] https://en.bitcoin.it/wiki/Atomic_cross-chain_trading.

have been included in the ledger and will revert to the transaction pool to be added into a later block.

Forces

- *Chain fork.* Chain fork may occur on a blockchain using Nakamoto consensus, like Bitcoin and Ethereum.
- *Frequency of chain fork.* Transaction handling and inter-block time differ significantly from one blockchain to another. A shorter inter-block time would lead to an increased frequency of forks.

Solution From the application perspective, one security strategy is to wait for a certain number (X) of blocks to be generated after the transaction is included into one block. After X blocks (1 inclusion block and X-1 confirmation blocks), the transaction is taken to be *committed* and thus perceived as immutable. The value of X can be decided by the developers of the blockchain-based applications, based on characteristics of the blockchain platform and the value or risk of the transaction. Figure 7.10 is a graphical representation of the pattern.

Consequences

Benefits:

- *Immutability.* The more blocks being generated after the block including the transaction, the higher probability of the immutability of the transaction.

Drawbacks:

- *Latency.* Latency between submission and commit of a transaction is affected by the consensus protocol, the inter-block time, and the number of confirmation blocks X. For example, this is around 1 h (10-min block interval with 6-confirmation) on Bitcoin. The larger value of the X, the longer the latency.

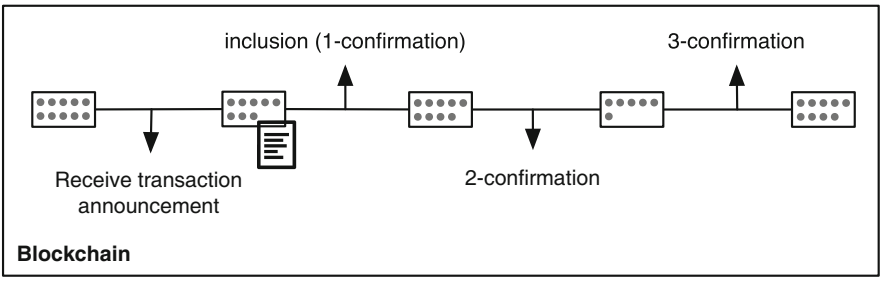


Fig. 7.10 X-Confirmation pattern. This work is based on an earlier work: Xu et al. (2018) © ACM, 2018. http://dx.doi.org/10.1145/3282308.3282312. Included here by permission

Related Patterns N/A

Known Uses

- *Bitcoin* users often choose 6-confirmation. The value 6 for the Bitcoin blockchain corresponds to the assumption that an attacker is unlikely to amass more than 10% of the total amount of computing power within Bitcoin network (measured by *hash rate*[34]) and that a negligible risk of less than 0.1% is acceptable.[35]
- *Ethereum* users sometimes recommend to choose 12-confirmation before assuming that a transaction is committed permanently with high probability.[36]

7.4 Contract Structural Patterns

This section discusses five smart contracts patterns. Essentially, smart contracts are programs running in transactions on a blockchain. Some of the design patterns and programming principles for conventional software environments are also applicable to smart contracts. If a public blockchain is used, the structural design of the smart contract has large impact on its execution cost. The cost of deploying a smart contract depends on the size of the smart contract(s) because the code is stored on blockchain, resulting in a data storage fee that is proportional to the code size. Thus, a structural design with more lines of compiled code costs more money. A consortium blockchain does not necessarily have tokens/cryptocurrency; therefore the monetary cost of smart contract deployment and execution is typically not a significant issue for consortium blockchains. However, blockchain size is still a design concern because the total size of the blockchain keeps growing as more blocks are appended to it and no block can ever be detached from it, and every full node stores a full replica of blockchain. Different structural designs of smart contracts may also affect performance.

7.4.1 Pattern 11: Contract Registry

Summary Before invoking a smart contract, the address of the latest version of the smart contract is located by looking up its name on a contract registry.

Context As with any software application, blockchain-based applications need to be upgraded to new versions. For instance, the on-chain functions defined in smart contracts need to be updated to fix bugs as well as to fulfil new requirements.

Problem Smart contracts deployed on blockchain cannot be upgraded because the code of the smart contracts is a type of data and data stored on a blockchain is immutable.

[34] https://blockchain.info/charts/hash-rate.

[35] https://en.bitcoin.it/wiki/Confirmation.

[36] https://blog.ethereum.org/2015/09/14/on-slow-and-fast-block-times/.

Forces

- *Immutability.* Every bit of data (including deployed smart contracts code) stored on a blockchain is immutable.
- *Upgradability.* There is a fundamental need to be able to upgrade all but short-lived applications and their smart contracts over time.
- *Human-readable contract identifier.* The identifier of a smart contract on blockchain platforms, like Ethereum, is a hexadecimal address, which is not human-readable.

Solution An on-chain registry contract is used to maintain a mapping between user-defined symbolic names and the blockchain addresses of the registered contracts. The address of the registry contract needs to be advertised off-chain. The creator of a contract can register the name and the address of the new contract to the registry contract after the new contract has been deployed. The invoker of a registered contract retrieves the latest address of the new smart contract from the registry contract. The corresponding functions provided by the registered contract can be upgraded by replacing the address of the old version contract in the registry contract with the address of a new version without breaking the dependency between the upgraded smart contract and other smart contracts that depend on its functions. The address of a contract is stored as a variable in the registry contract. The value of contract variables can be updated. The registry contract can have a permission control module to maintain write permissions. Note that all the previous values of the variable are still stored in the blockchain history. Figure 7.11 is a graphical representation of the pattern.

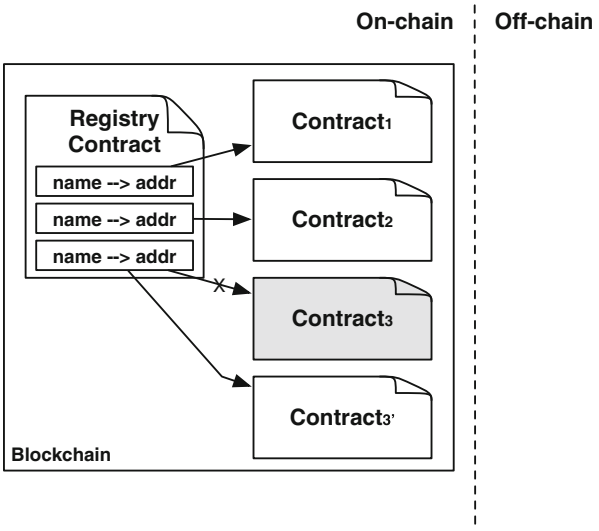


Fig. 7.11 Contract registry pattern. This work is based on an earlier work: Xu et al. (2018) © ACM, 2018. http://dx.doi.org/10.1145/3282308.3282312. Included here by permission

Consequences
Benefits:

- *Human-readable contract name.* The registry contract maintains a mapping between human-readable names and the hexadecimal addresses of the smart contracts. A human-readable form of smart contract names may be desired, for example, to be exposed to the user interface. A human-readable name is also useful for developers.
- *Constant contract name.* The smart contract associated with a registered name can be updated without changing its name. This way dependencies relying on the name of the smart contract do not get broken.
- *Transparent upgradability.* The smart contract associated with a registered name could be replaced by a new version without breaking the dependencies based on the human-readable name.
- *Version control.* Version control can be integrated in the registry contract to allow a lookup based on the name and version of a smart contract. Old versions of a smart contract that are no longer needed should be terminated.

Drawbacks:

- *Limited upgradability.* Upgradability is still limited if the functions defined in the smart contract are directly invoked by other contracts. Although the implementation of the function can be upgraded, the interface (i.e. function signature) cannot be modified without breaking the link to dependent smart contracts. Similar methods as for API/service interface management need to be implemented, e.g. through versioning and depreciation flags.
- *Cost.* There is an additional cost to maintain a registry that contains the mapping between the contract names and their addresses. Furthermore, all inter-contract function calls require a registry lookup to find the latest version of the smart contract to be invoked.

Related Patterns *Embedded permission* (Section 7.4.3) can be used for write permissions. *Data contract* (Section 7.4.2) and this pattern can work together to further improve upgradability of smart contracts.

Known Uses

- ENS[37] is a name service on Ethereum blockchain, which is implemented as smart contracts. ENS maintains a mapping between both smart contracts on-chain and resources off-chain and simple, human-readable names.
- ENS can be viewed as a contract registry built in a blockchain platform, which is accessible to everyone. A blockchain-based application can also maintain a separate registry contract for the application.

[37] https://ens.domains.

- Regis[38] is an in-browser application that makes it easy to build, deploy, and manage registries as smart contracts on Ethereum. It allows user-defined key-value pairs. It can be used to create a contract registry.

7.4.2 Pattern 12: Data Contract

Summary Store data in a separate smart contract.

Context Many blockchain-based applications must be upgraded over time. In general, the logic and the data that form part of the application on the blockchain may change at different times and with different frequencies. There are different ways to store a data on blockchain, as discussed in *hash integrity* pattern (Section 7.2.3).

Problem Storing data on blockchain is expensive, and there is a limitation on the amount of data and amount of computation a transaction can contain. In the context of upgrading smart contracts, the upgrading transactions might contain a large data storage for copying the data from the old version of the smart contract to the new version of the smart contract. Porting data to a new version might even require multiple transactions, e.g. when the block gas limit on Ethereum prevents an overly complex data migration transaction.

Forces

- *Coupling*. Smart contracts can live forever on blockchain if they are not explicitly terminated. If a smart contract is deactivated in this way, the data stored in the smart contract cannot be accessed through the smart contract functions any more—although it can still be accessed externally with some effort, e.g. for provenance or audit purposes.
- *Upgradability.* Many applications need to be able to be upgraded over time.
- *Cost.* If a public blockchain is used, storing data on blockchain costs money. Copying data from an old version of a smart contract to a new version should be avoided or minimized.

Solution To avoid moving data during upgrades of smart contracts, the data store is isolated from the rest of the code. In the context of blockchain, data could be separately stored in different smart contracts to enable isolation. One example of a generic data structure is a mapping to store *SHA3* key and value pairs. The keys are used in lieu of variable names. Figure 7.12 is a graphical representation of the pattern.

[38] https://regis.nu/.

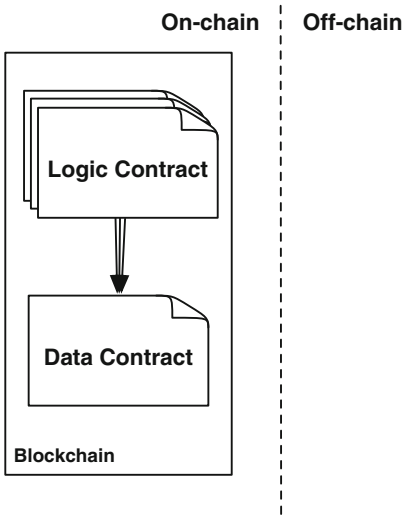


Fig. 7.12 Data contract pattern. This work is based on an earlier work: Xu et al. (2018) © ACM, 2018. http://dx.doi.org/10.1145/3282308.3282312. Included here by permission

Consequences
Benefits:

- *Upgradability.* By separating data from the rest of the code, the logic of the application can be upgraded without affecting the data contract.
- *Cost.* Since the data is separated from the rest of the code, there is no cost for migrating data when the application is upgraded.
- *Generality.* If the data can be cleanly separated and generalized, there would be an additional benefit: the generic data contract can be used by all related logic smart contracts.

Drawbacks:

- *Cost.* If a public blockchain is used, storing a piece of data in a generic data structure costs more money than a strictly defined data structure. For example, a mapping between *SHA3* key and value pairs will use more memory than a more strictly defined data structure that does not store key names. Querying the data is also more indirect. This is the cost of a generalized solution.

Related Patterns *Contract registry* (Section 7.4.1) and this pattern can work together to further improve upgradability of smart contracts.

Known Uses

- *ChronoBank*[39] is a blockchain project that tokenizes labour and provides a market for professionals to trade their labour time with businesses. It uses a smart contract with a generic data structure as the data store used by all the other logic smart contracts.

[39] https://chronobank.io/.

- *Colony*,[40] a platform for open organizations running on Ethereum. Similar to ChronoBank, Colony has a data contract with a generic data structure.

7.4.3 Pattern 13: Embedded Permission

Summary Smart contracts use embedded permission control to restrict access to the invocation of the functions defined in the smart contracts.

Context All smart contracts running on a blockchain can be accessed and called by any blockchain participants or other smart contracts by default. There are no privileged users, and, in the case of public blockchain, anyone can join the network to access all the information and code stored and running on blockchain.

Problem A smart contract by default has no owner, meaning that once deployed the author of the smart contract has no special privilege to invoke on the smart contract. A permission-less function can be triggered by unauthorized users accidentally or maliciously. Such a permission-less function can be a vulnerability for a blockchain-based application. For example, a permission-less function discovered in a smart contract library used by the Parity multi-sig wallet caused the freezing of about 500K Ether.[41] In 2016, seven percent of smart contracts on the public Ethereum blockchain could be terminated without authority.

Forces

- *Security.* The functions defined in the smart contracts should be only callable by authorized participants. Due to the transparency of public blockchains, all smart contracts are also publicly available. In contrast, in a conventional software system, the internal logic is normally not visible to end users. Interaction with the software system is either through a user interface or API, where it is possible to enforce access control policies.

Solution Add permission control to every smart contract function to check permissions for every caller that triggers the functions defined in the smart contract. Permission is determined based on the blockchain addresses of the caller. This can be done by checking the authorization of the caller before executing the logic of the function: unauthorized calls are rejected and the execution of the function terminated before reaching the core logic of the function. Figure 7.13 is a graphical representation of the pattern.

[40] https://colony.io/.

[41] https://paritytech.io/a-postmortem-on-the-parity-multi-sig-library-self-destruct/.

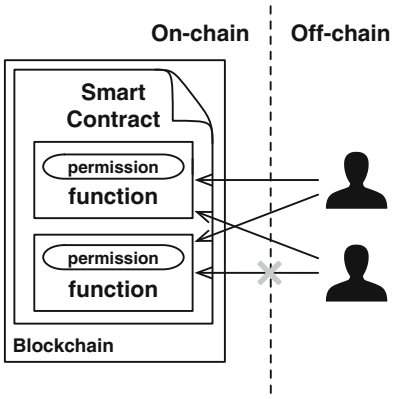


Fig. 7.13 Embedded permission pattern. This work is based on an earlier work: Xu et al. (2018) © ACM, 2018. http://dx.doi.org/10.1145/3282308.3282312. Included here by permission

Consequences
Benefits:

- *Security.* Only the participants and smart contracts that are authorized by the smart contract can call the corresponding functions successfully.
- *Secure authorization.* Authorization is implemented in smart contracts running on blockchain, which leverages the properties provided by blockchain.

Drawbacks:

- *Cost.* On a public blockchain, extra code that implements the permission control mechanism also has additional deployment and runtime cost.
- *Lack of flexibility.* Permissions are defined in the smart contract before its deployment; therefore they are difficult to change. However, permissions may be required to be dynamic. A mechanism is needed to support dynamic granting and removal of permissions.

Related Patterns *Multiple authorization* (Section 7.3.1) and *off-chain secret enabled dynamic authorization* (Section 7.3.2) are different ways to design authorization.

Known Uses

- The Mortal contract discussed in the Solidity tutorial[42] restricts the permission of invoking the *selfdestruct* function to the 'owner' of the contract—where 'owner' is a variable defined in the contract code itself.
- The *Restrict access* pattern suggested in the Solidity tutorial[43] uses *modifier* to restrict who can make modifications to the state of the contract or call the functions of the contract. *Modifier* is a mechanism to add a piece of code before the function to check certain conditions. *Modifier* can make such restrictions highly readable.

[42] http://solidity.readthedocs.io/en/develop/contracts.html.
[43] http://solidity.readthedocs.io/en/develop/common-patterns.html.

7.4.4 Pattern 14: Factory Contract

Summary An on-chain template contract is used as a factory that generates contract instances from the template.

Context Applications based on blockchain might need to use multiple instances of a standard contract with customization. Each contract instance is created by instantiating a contract template. For example, in a business process management system, each of the business process instances might be represented by a smart contract being generated from a contract template representing the business process model. The template can be stored off-chain in a code repository, or on-chain, within its own smart contract.

Problem Keeping the contract template off-chain cannot guarantee consistency between different smart contract instances created from the same template because the source code of the template can be independently modified.

Forces

- *Dependency management.* Storing the source code of a smart contract off-chain in a code repository introduces the issue of integrating more systems into the blockchain-based application.
- *Secure code sharing.* Blockchain can be used as a secure platform to share code of smart contracts. As opposed to a traditional code repository, changes of code deployed on a smart contract can be strictly limited or prohibited.
- *Deployment.* If a public code repository, like GitHub, is used to store the source code of a smart contract, a component is needed to implement the function of deploying smart contracts on blockchain, otherwise the end users need to understand how to deploy smart contracts by sending transactions with the customized source code of the contract definition.

Solution Smart contracts are created from a contract factory deployed on blockchain. The factory contract is deployed once from the off-chain source code. The factory may contain the definition of multiple smart contracts. Smart contract instances are generated by passing parameters to the contract factory to instantiate customized smart contract instances. A factory contract is analogous to a *Class* in an object-oriented programming language. Every transaction that generates a smart contract instance instantiates an *object* of the factory contract class. This contract instance (the object) will maintain its own properties independently of the other instances but with a structure consistent with its original template. Figure 7.14 is a graphical representation of the pattern.

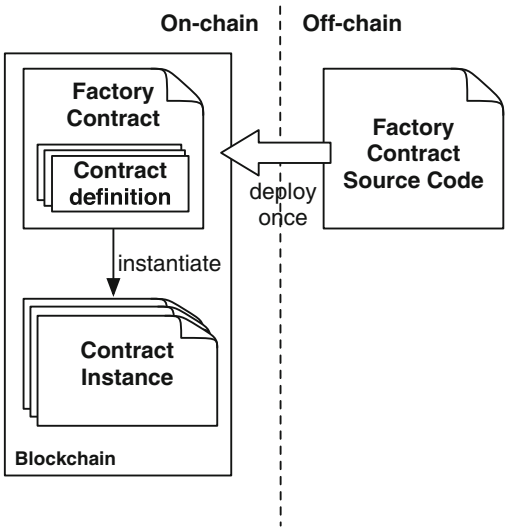


Fig. 7.14 Factory contract pattern. This work is based on an earlier work: Xu et al. (2018) © ACM, 2018. http://dx.doi.org/10.1145/3282308.3282312. Included here by permission

Consequences
Benefits:

- *Security.* Keeping the factory contract on-chain guarantees the consistency of the contract definition.
- *Efficiency.* If the contract definition is kept on-chain in a factory contract, smart contract instances are generated by calling a function defined in the factory contract.

Drawbacks:

- *Deployment cost.* If a public blockchain is used, using factory contract requires extra cost to deploy the factory contract.
- *Function call cost.* If a public blockchain is used, creating a new smart contract instance requires extra cost to call a function defined in the factory contract.

Related Patterns *Contract registry* (Section 7.4.1). A contract registry can be used to store the addresses of all the smart contract instances generated from a factory contract. The factory and instance registry can be implemented in the same contract, although that limits upgradability.

Known Uses

- A tutorial from Ethereum developers[44] about how to create a contract factory from which smart contract instances can be created.
- The factory pattern has been applied in a real-world blockchain-based healthcare application.

[44] https://ethereumdev.io/manage-several-contracts-with-factories/.

- The business process management system in an academic work uses a contract factory to generate process instances.

7.4.5 *Pattern 15: Incentive Execution*

Summary A reward is provided to the caller of a contract function for invoking it.

Context Smart contracts are event-driven programs, which cannot execute autonomously. All the functions defined in a smart contract need to be triggered by a transaction either from an external account or from another smart contract to execute. Other than the functions that provide regular services to users, some functions need to run asynchronously from regular user interaction, for example, to clean up expired records or make dividend payouts, etc. Such functions usually involve a time, after which the function should start.

Problem Users of a smart contract have no direct benefit from calling accessory functions. If a public blockchain is used, executing these functions causes extra monetary cost. Some accessory functions are expensive to execute.

Forces

- *Completeness.* The regular services provided by a smart contract are supported by some accessory functions.
- *Cost.* Execution of accessory functions causes extra cost to users.

Solution Reward the caller of a function defined in a smart contract for invoking the execution, for example, sending back a percentage of payout to the caller to reimburse the (gas) execution cost. Figure 7.15 is a graphical representation of the pattern.

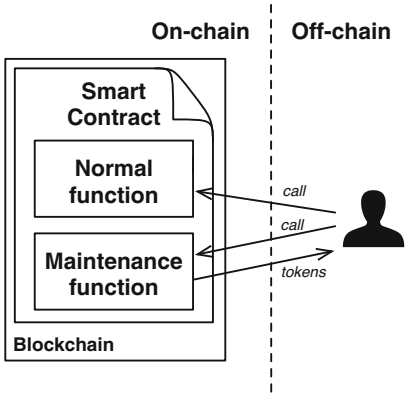


Fig. 7.15 Incentive execution pattern. This work is based on an earlier work: Xu et al. (2018) © ACM, 2018. http://dx.doi.org/10.1145/3282308.3282312. Included here by permission

Consequences
Benefits:

- *Completeness.* The execution of the accessory function helps to complete the regular services provided by the smart contract.
- *Cost.* Users who expend resources to execute the accessory functions are compensated by the reward associated with the execution.

Drawbacks:

- *Unguaranteed execution.* Execution cannot be guaranteed even with incentives. Thus, another option is to embed the logic of accessory functions into other regular functions that users have to call to use the services.

Related Patterns N/A

Known Uses

- *Regis*[45] is an in-browser tool for developers to create smart contracts representing registries on Ethereum. The functions that clean up the expired records provide incentives for users to execute them.
- *Ethereum alarm clock*[46] is a service provided by a smart contract running on Ethereum. It facilitates scheduling function calls for a specified block in the future and provides incentive for users to execute the scheduled function.

7.5 Summary

Blockchain can be used as a core component of (possibly large-scale) decentralized software systems. For effective use of blockchain to this end, patterns can convey means to make good use of blockchain in the design of systems and applications. In this chapter, we present a pattern collection for blockchain-based applications. Our pattern collection includes three patterns about interaction between blockchain and the external world, four data management patterns, three security patterns, and five contract structural patterns. The pattern collection provides architectural guidance for developers to build applications on blockchain. Some patterns are designed specifically for blockchain-based applications considering the unique properties of blockchain. Others are variants of existing software patterns applied to smart contracts.

[45] https://regis.nu/.
[46] http://www.ethereum-alarm-clock.com/.

7.6 Further Reading

This chapter is partly based on our earlier works (Xu et al. 2018).

In software engineering, a design pattern is a reusable solution to a problem that commonly occurs within a given context during software design. A definition and a formalization of design patterns are given in Beck and Cunningham (1987) and Meszaros et al. (1998).

A few other design patterns of blockchain-based applications or smart contracts can be found in the literature. Bartoletti and Pompianu (2017) conduct an empirical analysis on smart contracts supported by different blockchain platforms. The paper focuses on the two most widespread ones, Bitcoin and Ethereum. Nine common programming patterns are identified in Solidity-based smart contracts by manually inspecting the publicly available source code. The identified programming patterns include tokens, authorization, oracle, randomness, poll, time constraint, termination, math, and fork check. Zhang et al. (2017) apply four existing object-oriented software patterns to smart contract programming in the context of a blockchain-based healthcare application. The applied software patterns include abstract factory, flyweight, proxy, and publisher-subscriber. Eberhardt and Tai (2017) propose five patterns for blockchain-based applications focusing on what data and computation should be on-chain and what should be kept off-chain, which include challenge response pattern, off-chain signatures pattern, content-addressable storage pattern, delegated computation pattern, and low contract footprint pattern.

The background of Ricardian contracts as one of the known uses is discussed in Grigg (2004). The details of the *Smart Contract Template* proposed by Barclays are discussed in Clack et al. (2016a,b). The logic-based language for smart contract definition can be found in Idelberger et al. (2016).

Chapter 8
Model-Driven Engineering for Blockchain Applications

with Alex Ponomarev and An Binh Tran

8.1 Introduction

Model-driven engineering is a methodology for using models at various levels of abstraction and for different purposes during software development. For some models the level of abstraction is low, so that the production code can be directly derived from the models. Other models use a high level of abstraction and only guide developers. Intermediate levels of abstraction can support model-based system analysis or might be used by system management tools. Depending on the purpose and the system, there can be various dimensions captured in models, from static structures (such as data models or deployment schemes) to dynamic aspects (like activity sequences). For code generation specifically, there are further options: code generation can be once-off, with subsequent evolution of the code independently of the model; or it can be repetitive, where the code is regenerated from the model following changes to the model. In the latter case, we can also distinguish one-way model-to-code code generation from round-trip code generation. In round-trip code generation, if the generated code is updated, the changes can be propagated back to the model level. This is an often desired but rarely achieved vision for model-based development.

In the context of blockchain-based applications, model-driven development is of particular relevance. First, code generation tools can implement best practices and well-tested building blocks, thereby avoiding code that contains common errors or is vulnerable to known attacks. Second, models can be independent of specific blockchain technologies or platforms, and code generation tools might cater for multiple target platforms. This can avoid lock-in to a specific blockchain platform and help application developers migrate to alternative technologies. Third,

X. Xu et al., *Architecture for Blockchain Applications*,
https://doi.org/10.1007/978-3-030-03035-3_8

models are often easier to understand than code. This can be particularly useful for communicating with business partners about smart contracts, and strengthen confidence in that code from all parties. Take the example of a contract that will hold funds in escrow and specifies conditions under which the funds will be paid. Such a smart contract is written by one party but used by others. All parties need to rely on the contract code working as expected (and not, say, transfer all funds held in escrow to its developer). It can be easier to verify the correctness of the model than the raw code, and tooling can ensure that the deployed code has not been changed after being derived from the model. Of course, the code generation tool also needs to be correct, but confidence in that can be established across many and varied uses of the tool.

We discuss two approaches for model-driven code generation in this chapter. The first uses process models for collaborative business processes that cross organizational boundaries. The second targets registries for assets, such as land titles, cars, or digital assets. It focuses on non-fungible assets, i.e. where the specific identity of an asset is important. For example, you probably care about *which* car you own, not just whether you own *one* (any) car. This is in contrast to fungible assets, such as shares in a company, where you care *how many* shares you own and where individual shares might not even be easily identifiable. For the latter, standards like ERC20[1] exist on Ethereum. Non-fungible assets often have asset-specific peculiarities that make model-driven development more useful.

8.2 Model-Driven Generation of Smart Contract Code for Collaborative Business Processes

8.2.1 Motivation

The integration of business processes, e.g. along the supply chain, has been found to contribute to better operational and business performance. A lack of trust, however, may hamper collaborative process performance. Once service-level agreements are in place, it can be a highly delicate question which partner should serve as a hub for controlling the collaborative process of several parties, or where a mediator process is hosted. While control asymmetries can be avoided by adopting a decentralized view (such as process choreographies) instead of central orchestration, it does not solve the general problem of trust in the control of the collaborative business process.

In this section, we describe how blockchain technology can address this lack-of-trust problem in collaborative business processes. More specifically, we describe an approach to map a collaborative process to a blockchain-based execution infrastructure that offers the following benefits. First, it provides a monitoring facility

[1] https://theethereum.wiki/w/index.php/ERC20_Token_Standard.

that integrates an automatic and immutable transaction history, which is useful for dispute resolution and even mandatory in some highly regulated industries. Second, smart contracts can be used as a direct implementation of the process control logic, specifically the mediator process that orchestrates the coordination between the involved parties. Third, the process logic can be enforced automatically, including payments, escrow, and conflict resolution.

8.2.2 Challenges of Collaborative Business Process Execution

We illustrate challenges of executing collaborative business processes by using an example supply chain scenario, shown in Fig. 8.1. The process starts with a Bulk Buyer placing an order with a Manufacturer. The latter calculates the demand and places an order for materials via a Middleman. This Middleman forwards the order to a Supplier and arranges transportation by a Special Carrier. Once the materials are produced, the Carrier picks them up at the Supplier site and delivers them to the Manufacturer. The Manufacturer produces the goods and delivers them to the Bulk Buyer. The process model as shown falls into the category of *choreography* since it is modelled from a global viewpoint and there is no party that sees all messages. A choreography is a global, participant-independent view of a collaborative process and focusses on the interaction points between different participants. This view only vaguely specifies what needs to be done by whom, but not how. In contrast, if all messages were sent and received by the Manufacturer, it could be modelled

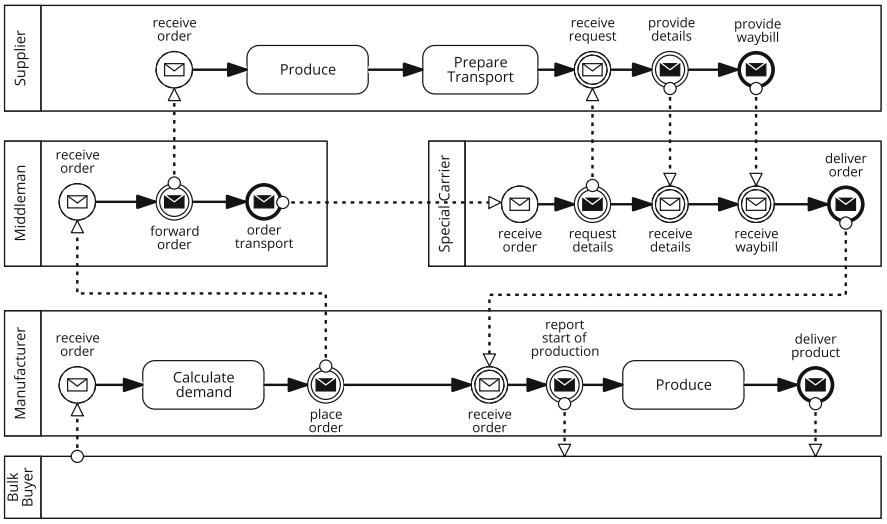


Fig. 8.1 Supply chain process example (Notation: BPMN). © 2016 by Springer International Publishing, part of Springer Nature, reprinted with permission

as an *orchestration* with the Manufacturer serving as a mediator. An orchestration is modelled from the viewpoint of a single party in a collaborative process and provides sufficient details to be executed on behalf of that party; other parties' activities are not specified in detail either.

This simple scenario already involves five participants. In case of delays and errors in the process, it would not be uncommon if the participants started blaming each other. Consider the case where the Manufacturer receives the materials 3 days later than agreed, with eight pallets being delivered instead of ten. The Supplier might argue that this is exactly in line with what was ordered by the Middleman, while the Middleman would claim the fault to be on the side of the Supplier. The situation could delicate for the Carrier if the Manufacturer refused to accept the delivery. The Carrier and the Manufacturer may be entitled to compensation from the Supplier or the Middleman, depending on who is responsible for the fault.

8.2.3 Blockchain-Based Collaborative Process Execution

In the following, we discuss a blockchain-based approach to address the lack-of-trust problem in collaborative business processes. A number of technical challenges arise during the adoption of blockchain for this purpose. As is the case for all blockchain-based applications, there is a cost (though not necessarily in cryptocurrency in private/consortium blockchains) for new transactions, computation, and data storage on blockchain platforms. As discussed throughout the book, not all aspects of collaborative processes should be dealt with inside smart contracts. Smart contracts cannot call external APIs outside the blockchain environment nor directly create blockchain transactions. This section describes how the approach addresses these challenges.

An overview of the approach is shown in Fig. 8.2. It uses blockchain to facilitate the collaborative processes in one of two ways:

(i) As a *choreography monitor*, a smart contract stores the process execution status of all participants by observing message exchanges. In this setting, the blockchain serves as immutable data storage to share process execution status and create an audit trail. Smart contracts check if interactions conform to the choreography model and enforce that model. In addition, the choreography monitor can manage automated payment points and escrow.
(ii) As an *active mediator* among the participants, it coordinates collaborative process execution. This includes all the above, as well as using smart contracts to drive the process execution and to implement data transformation, checking of conditions, and calculations.

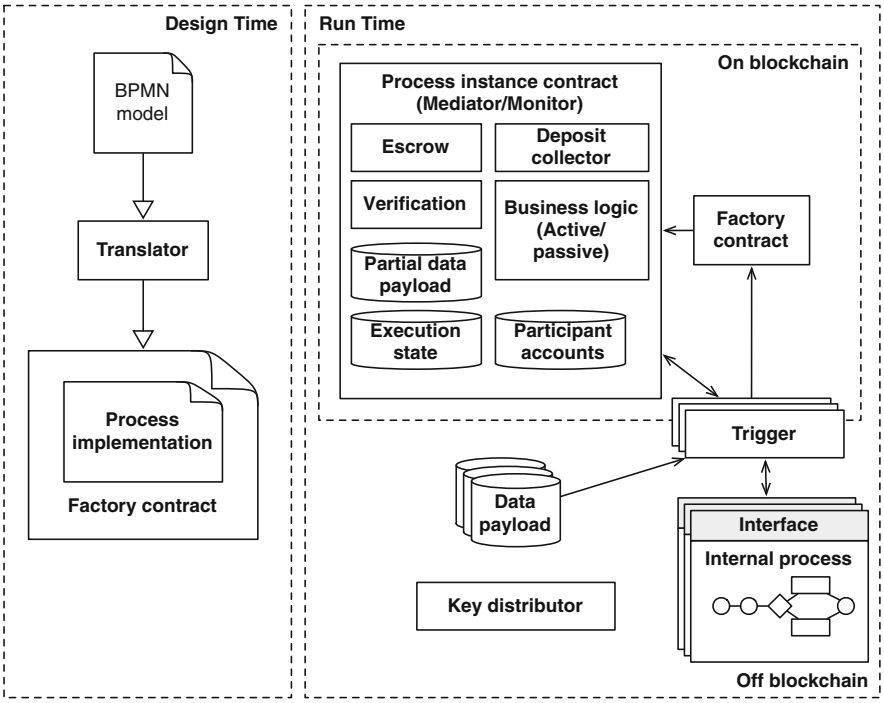


Fig. 8.2 Overview of the process model-based approach. © 2016 by Springer International Publishing, part of Springer Nature, reprinted with permission

These options are supported by the following main components:

- At design time, a **translator** derives from a process specification described in, e.g. Business Process Model and Notation (BPMN),[2] a smart contract in a programming language (such as Solidity, the language used by Ethereum). The generated smart contract is a factory for mediators or choreography monitors and as such implements the factory contract patterns discussed in Section 7.4.4.
- For Option (i), a **choreography monitor** or **C-Monitor** uses smart contracts *to monitor* the collaborative business processes. The C-Monitor is split into a factory and case-specific C-Monitor instances. All of them are smart contracts. The factory instantiates the case-specific monitors as needed and contains the blueprint for C-Monitor instances. The C-Monitor instance tracks the interactions of a choreography instance and combines them into a consolidated view of the current state of the execution. Optionally, it can trigger automatic conditional payment from escrow, when certain points in the choreography are reached.

[2] http://www.bpmn.org/.

- For Option (ii), an active **mediator** uses a smart contract to *implement* the collaborative business processes. As with the C-Monitor, it is split between a factory and a set of instances and offers a consolidated view of the process state. In contrast to the C-Monitor, the mediator always plays an active role, receiving and sending messages according to the business logic defined in the process model. It also may transform data or execute other computations.
- **Interfaces** or **triggers** connect the process executing on blockchain and the external world. Because smart contracts cannot directly interact with the world outside the blockchain, a trigger plays the role of an organization's agent. It holds confidential information and runs on a full blockchain node, keeping track of the execution context and status of running business processes. The trigger calls external APIs if needed, receives API calls from external components, and updates the process state in the blockchain based on external observations. It further keeps track of data payload in API calls and keeps the data in an external database when appropriate.

With these components, the approach ensures that (1) participants can execute collaborative processes over a blockchain network of untrusted nodes; (2) the state only progresses when messages (in the form of transactions) are received that are expected at the current execution state of the process, and only if they come from the correct party (else they are rejected); (3) payments and escrow can be coded into the process; and (4) the immutable blockchain ledger keeps a log of all transactions, successful or not. Next, we explain the above components in more detail.

Design Time: Translator

The translator is used at design time: it takes an existing business process specification as input and generates a corresponding factory smart contract, which implements the C-Monitor or mediator and can be deployed and executed on the blockchain.

In a collaborative process, the complete functionality must be split and distributed between the smart contract and the triggers. The translator creates the artefacts in such a way that the triggers and the smart contract collaborate directly with each other over the blockchain network. The smart contract contains all on-chain code, and the triggers connect enterprise systems, UIs, and other external components to the contract and vice versa.

When the translator is called, it is often not known which participants will play which roles. Also, organizations may want to execute many instances or cases of a process over time. Therefore, the translator outputs a **factory contract**, which in turn contains all information needed for instantiating the process. In addition to the factory contract, the translator can output an interface specification per role (e.g. buyer, manufacturer, and shipper) in a collaborative process, to be distributed to the respective triggers. The factory contract includes a method for instantiation, which, if invoked, creates a **process instance contract**. The process instance contract

contains the implementation of the business logic and takes the form of a C-Monitor or mediator, depending on the content of the original process specification and how it was translated.

The process instance contract (see Listing 8.1) is generated from the business logic which the translator inserted into the factory contract. The process instance contract consists of a list of storage variables that represent the execution state of the process instance. To optimize the cost, we could further minimize the size of the data stored on chain, but that lowers readability of the code, and we therefore include the less optimized version here. Two types of elements in a business process are implemented as functions in Solidity, namely, tasks and AND-Join gateways. Tasks are called by triggers, gateways only internally within the smart contract and therefore can be marked as private functions.

To start the process execution, the first task is activated in the constructor function `ProcessMonitor()`. To execute the task 'order goods', the function `Order_Goods()` is called in a blockchain transaction, e.g. through an invocation of the trigger. The corresponding function in the code first checks if the task that has been called is activated and whether it was called by the right participant (`msg.sender == participants[0]`); if not, the call is aborted with `return false`. Otherwise, task-specific code is executed, and finally the execution state of the process is advanced by updating the activation variables. The technology can handle complex processes with various types of gateways, details of which we omit here.

The task activation variables define the execution state of the process instance. Each task invocation includes steps that implement *process enforcement*: The call is accepted only if the call conforms to the process model in its current state and only if the call is made by the participant that is assigned to the role that is supposed to execute the task. Otherwise, it returns false to indicate that the execution has not succeeded, which can be interpreted as an alert to all participants. The data (e.g. a message) included in the function call is forwarded, as a smart contract log entry (not shown in the code). Payments (direct or to/from escrow, relating to cryptocurrencies or tokens) can be associated with tasks, in which case they are performed on-chain. Computational tasks, e.g. for data transformation, could be performed on-chain or off-chain depending on cost analyses.

After generating a smart contract, the translator can also calculate the gas cost estimates for executing the smart contract. This serves as an indication of the cost to execute process instances in the blockchain, which can in turn be used for budgeting and/or capacity planning, depending on the blockchain configuration to be used at runtime. For more details on cost estimates, see Chapter 9.

Runtime Environment: Executing Processes as Smart Contracts

The translator generates all artefacts needed for runtime execution. We start their description with C-Monitors, which allow passive monitoring of choreographies with optional escrow. Active mediators can be seen as an extension of C-Monitors,

```
contract ProcessMonitor {
    address[] participants;

    // ---------- process variables
    bool taskOrderGoodsActivated = false;
    bool taskPlaceOrderForSuppliesActivated = false;
    ...

    function ProcessMonitor(address[] _participants) {
        taskOrderGoodsActivated = true;
        ...
    }

    function Order_Goods(...) returns(bool) {
        if ((taskOrderGoodsActivated  && msg.sender ==
            participants[0]) {
            // task-specific code
            ...
            // update execution state
            taskOrderGoodsActivated = false;
            taskPlaceOrderForSuppliesActivated = true;
            return true;
        }
        return false;
    }
    ...
}
```

Listing 8.1 Example of C-Monitor contract code in Solidity, © 2016 by Springer International Publishing, part of Springer Nature, reprinted with permission

and the additional functionality is explained next. The third important concept for runtime, the triggers, and the interaction between triggers and smart contracts are covered afterwards. Finally, we describe how technical challenges like key distribution and encryption are handled.

Choreography Monitor The first way of facilitating collaborative processes is to use a smart contract as a C-Monitor, with optional escrow and conditional payment at certain points of the processes. How the private processes of participants are executed within their regular enterprise systems is largely out of scope here; however, the assumption is that they can make API calls (to their respective triggers) for coordination. Of course the internal enterprise systems can be extended to directly incorporate the triggers as well.

For a new process instance, an instance contract is generated from the factory contract. Initialization includes registering participants and their public keys (account addresses) to roles. This enables the instance contract to ensure authentication, e.g. such that the goods can only be ordered by the Bulk Buyer in our running example process. The C-Monitor instance contract contains variables for storing the role assignment and for the process execution status, as shown in Listing 8.1.

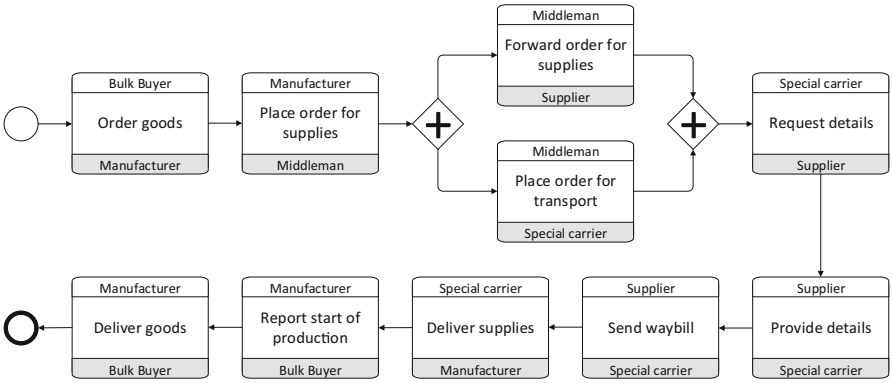


Fig. 8.3 BPMN choreography diagram of the process in Fig. 8.1 (Notation: BPMN). © 2016 by Springer International Publishing, part of Springer Nature, reprinted with permission

During execution, the participants do not interact with each other directly. Instead, they invoke functions on the instance smart contract to exchange messages as data payloads. The contract checks if a message is a transaction signed by the correct participant and that the message is permitted at the current state of the process. It then writes the result into the smart contract event log. The log is analysed by all triggers, which react upon observing relevant information. With this mechanism, participants exchange the messages and simultaneously advance the state of the collaborative process.

Consider the choreography in Fig. 8.3, which is a representation of the collaborative process from Fig. 8.1. All tasks are communication tasks between roles. The C-Monitor is used to exchange messages, to check conformance with the choreography model, and to track the status. While triggers and smart contracts together forward messages and update the state of the process, the state can also be inferred from the raw blockchain data. In this way, conformance checking is done implicitly by the C-Monitor, and all transactions (successful or not) are logged in the blockchain. The handling of escrow is described below.

As discussed throughout the book, a main design decision for blockchain-based systems is about which parts of computation and data should be on-chain and which should be off-chain. The blockchain provides neutral territory to verify computational results and provide agreement on transactions' outcomes, but the amount of computational power and data storage space available on the network remains limited, even in non-public settings. The computational power and data storage space on public blockchains incur monetary costs. If the input/output data payload is sizeable, it should likely be stored off-chain. In this case, we can use the off-chain data storage pattern (Section 7.2.3): the transactions include a URI of the input/output data payload and its hash value on the blockchain. The data can then be retrieved from the URI, and the hash allows verification of the integrity of the data.

Mediator The second way of facilitating collaborative processes is to use the smart contract as an active mediator. This orchestrates calls between different organizations. Like the C-Monitor, the mediator is implemented as factory and instance contracts. The instance contracts use the same components as the C-Monitor instances, including registration of involved participants and their roles, information specific to a process instance, and escrow. Mediators also implement active components, to transform data and to receive and send messages and payments.

While message and payment handling are straightforward to achieve using smart contracts, data transformation can easily become uneconomical. In this case, we can apply the oracle pattern from Section 7.1.1 as follows. One of the triggers can be designated to be called from the mediator, transform the data, and send a message with the output back to the mediator. The smart contract can also require that multiple triggers agree on the result of the computation, either using multi-signatures or using a separate transaction from each trigger confirming the result.

Triggers A blockchain is a closed environment, where the deployed smart contracts cannot directly call external APIs. In the approach discussed here, a trigger (or blockchain interface) can connect the participants' internal processes with the blockchain. It monitors the process execution status, logically receives messages from smart contracts and calls external APIs, or receives API calls and logically sends messages to smart contracts accordingly.

Triggers are programs running on full nodes of the blockchain network. Triggers can be distributed on multiple full nodes for increased reliability. In the typical setup, every participant operates its own trigger deployed on a node it controls, and the participant's internal systems only communicate with their own triggers. Since a trigger is required to hold private keys for the participants on whose behalf it operates, a high degree of trust in the individual trigger is required. Normally each participant should operate its own trigger.

When a new process instance is created, the participants register their roles and public keys. Recall that the public key corresponds to the account address of a participant. All keys and role assignments are passed to all triggers associated with the process instance, so everyone knows which role is played by whom and can verify messages accordingly. With the private key it holds, the trigger can encrypt or sign a message, allowing the contract and the other participants to verify its messages. In this fashion, it can also create payment transactions using cryptocurrency held in the accounts it controls.

During process execution, the trigger is receptive to API calls from its owner, as well as to logical messages from the process instance contracts. The interaction between internal process implementations, triggers, and the process instance smart contract is shown in simplified form in Fig. 8.4. When a trigger's API is called from its owner, the trigger translates the received message into a blockchain transaction, test-calls the smart contract locally, and if that is successful sends the transaction to the instance contract. The local test call allows the trigger to check if the choreography task that expects this message is activated. If not, the local test call will return false, and the trigger will know the smart contract is not in a state where

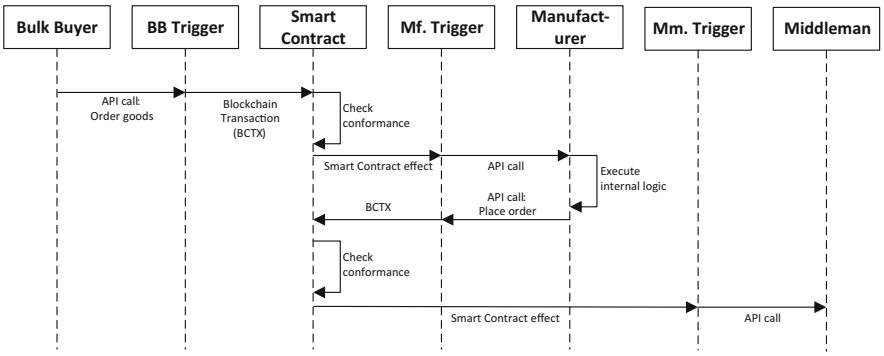


Fig. 8.4 Sequence diagram for the first two tasks in Fig. 8.3. © 2016 by Springer International Publishing, part of Springer Nature, reprinted with permission

the message can be sent. In turn, the trigger can alert its caller or delay the message and retry periodically. Note that, even if the local test call is successful, the real transaction can still fail, e.g. if the status has been updated between the test call and the transaction being processed. When the trigger receives a logical message from the instance contract, it updates its local state and makes an API call to the internal enterprise application that implements the private process for its owner.

Finally, the trigger can take care of sizeable data payloads. For incoming API calls, it moves the data to secure storage, hashes it, and attaches a URI and the hash to the outgoing transaction. For incoming messages from the blockchain, it retrieves the data via its URI, checks if the hash matches, and sends it on to the internal process implementation.

Encryption and Key Distribution All information on a blockchain is accessible to all nodes within the network. We store two types of information on blockchain, namely the process execution status and the data payload (or its URI/hash). To preserve the privacy of the participants, we have the option to encrypt the data payload before inserting it into the blockchain. However, the process execution status is not encrypted because the C-Monitors and mediators need to process this information. Encrypting the data payload means that mediators cannot perform data transformation at all, but can resort to the source participant's trigger for this task.

We assume the participants exchange their public keys with each other before a process instance is initiated by one of the participants. Thus, the key distribution is handled off-chain. Since participants need to find each other through off-chain mechanisms before starting a collaborative process, this typically does not introduce much overhead.

Encrypting data payload for all process participants can be achieved as follows. One participant creates a secret key for the process instance and distributes it during initial key exchange. When a participant adds data payload to the blockchain, it first symmetrically encrypts this information using the secret key. Thus, the publicly accessible information on blockchain is encrypted, i.e. useless to anyone who has

no access to the secret key. The participants involved in the process instance have the secret key and can decrypt the information.

Encrypting data payload between two process participants, in contrast, may be desired if two participants want to exchange information privately through the process instance. For this case, the sender can asymmetrically encrypt the information using the receiver's public key; only the receiver can decrypt it with its private key.

Escrow The C-Monitor or mediator can also work as an escrow for conditional payment at designated points. Similar to an escrow agent, e.g. in real estate transactions, the smart contract receives money from one or more parties, and only releases the money to other parties once certain criteria are met. For the receivers this has the benefit that they can observe that the money is actually there before doing work; and the sender does not have to pay upfront, trusting it will eventually receive the goods or service in return.

In the running example process, the Manufacturer needs to pay the Middleman, Supplier, and Carrier when it receives the goods. But the Supplier is unwilling to send the goods without some guarantee that it will get paid. Therefore, the Manufacturer puts the money in escrow, namely the account of the process instance contract, when ordering the goods. This account is exclusively controlled by the smart contract code, but the presence of the funds in escrow is visible to everyone on the blockchain (including the Supplier). Later, both the Carrier and the Manufacturer confirm the delivery of the goods, which triggers automatic payment from the escrow account to the Middleman, Supplier, and Carrier.

The smart contract defines under what conditions the money can be transferred and how the money should be transferred. Thus, when a payment function is triggered, the smart contract automatically checks the defined conditions, and transfers the money according to the defined rules. It is, however, of high importance to specify rules that cover all possible scenarios and the respective outcomes: e.g. what shall happen with money in escrow if the Manufacturer and the Carrier disagree about the delivery of the goods or their condition? Implementing the rules in a smart contract does not prevent possible conflicts, but it allows their automatic enforcement.

8.2.4 Discussion

Conflict Resolution Following up on the *conflict example* from Section 8.2.2, we discuss how conflict resolution can be implemented in our approach. Recall that there was disagreement about the amount of supplies ordered. The blockchain inherently provides an immutable audit trail. Thus it is trivial to review the original order and waybill messages, and the culprit can be identified through such inspection. Say that the Supplier was at fault, but the Manufacturer paid crypto-coins into escrow. How does it get its money back? The conditions for reimbursement

from escrow need to be specified in the smart contract, but then they can be invoked at a later time. For instance, the participants may agree upfront that the Manufacturer gets reimbursed only if the Middleman agrees to that; then the Middleman sends a transaction to that effect, and the Manufacturer's money is transferred back to its account.

Trust Blockchain provides a trustworthy environment, without requiring trust in any single entity. In contrast, in the traditional model participants who do not trust each other need to agree on a third-party which is trusted by all. Blockchain can replace this trusted third-party. This is of particular interest in cases of *coopetition*, i.e. organizations cooperate for specific cases to achieve business goals that are mutually beneficial, but compete in other cases. In such cases, it is important that the entity which executes the joint business process is neutral. Say, *Org1*, *Org2*, and *Org3* are in coopetition but want to have a joint process to achieve some business goal. However, *Org1* would not accept *Org2* or *Org3* to control the process, and neither of those would accept *Org1*. Using the blockchain for process execution enables trustless collaboration, as it is not controlled by a single entity. The translator allows the deployment of business processes on a blockchain network without the need to manually implement the corresponding smart contract.

Trust in the deployed bytecode for a process can be established as follows: each participant has access to the process model, translates it to Solidity with the translator, and uses an agreed-upon Solidity compiler. This results in the same bytecode, and each participant can verify that the bytecode deployed on the blockchain has not been manipulated. Finally, the *trigger* allows for seamless integration into service-based message exchanges. However, each trigger is a fully trusted party, and by default each organization should host its own trigger.

Privacy Public blockchains do not guarantee data privacy: anyone can join a public blockchain network without permission, and information on the blockchain is public. Thus, for scenarios like collaborative process execution, a permissioned blockchain may be more appropriate, configured so that joining it requires explicit permission. Even with permission management, the information on blockchain is still available to all the node operators on the blockchain network. While we discussed a method to encrypt the data payload of messages, the process status information is available to all nodes. As such, if *Org1*'s competitor, *Org4*, knows which account address belongs to which participant, it can infer with whom *Org1* is doing business and how frequently. This can be mitigated by creating a new account address for each process instance: the space of addresses is huge and account creation trivial. However, this method prevents building a reputation, at least on the blockchain.

Off-Chain Data Store As discussed above, for large data payloads the off-chain data storage pattern from Section 7.2.3 can be used: only metadata with a URI and a hash is stored on-chain, and the actual payload data is kept off-chain—accessible with the URI.

8.2.5 *Conclusion*

Collaborative process execution is problematic if the participants involved have a lack of trust in each other. In this section, we discussed using blockchain and its smart contracts to circumvent the traditional need for a centralized trusted party in a collaborative process execution. First, a translator can translate process models into smart contracts that can be executed on a blockchain. Second, the approach utilizes the computational infrastructure of blockchain to coordinate business processes. Third, to connect the smart contracts on blockchain with enterprise systems and the external world, we discussed the concept of triggers. A trigger converts API calls to blockchain transactions directed at a smart contract and receives status updates from the contract that it converts to API calls. Triggers can thus act as a bridge between the blockchain and an organization's private process implementations. Additional benefits of this approach include the option to build escrow and automated payments into the process and that the blockchain transactions from process executions form an immutable audit trail.

8.3 Model-Driven Registry Generation for Blockchain

In this section, we discuss model-driven development of registries for assets, such as land titles, cars, or digital assets. A registry is a list of information recorded and managed by a trusted authority. For example, a government might maintain a registry to store information about businesses, including their business number and name. Usually registries are operated as a centralized service, but this creates a single point of failure for the whole system. One approach to address this limitation is to use blockchain and smart contract technologies. As explained in the beginning of the chapter, we focus here on non-fungible assets, where the identity of the individual asset is important. Fungible assets like interchangeable tokens can instead make use of well-proven standards like ERC20, and are not discussed here.

Building registries on a blockchain can provide increased confidence in data integrity, availability, transparency, and immutability, and there is strong interest from industry and government around this idea. In particular, data integrity and availability are two of the key requirements of registries. Additionally, if we use a blockchain as a unified infrastructure, multiple registries can more easily interact with each other. There are registries being built on blockchain in ad hoc ways, for example, Namecoin,[3] which is a domain name registry that shares the same network with Bitcoin, and Ascribe,[4] which is an artwork registry that allows artists to register

[3] https://namecoin.org/.

[4] https://www.ascribe.io/.

and manage the ownership of their digital artwork. However, building a registry on blockchain is non-trivial, and the code needs to be of very high quality since it typically manages assets of value.

In this section, we discuss *Regerator*, which is a tool that follows a model-driven approach to provide templates for developers to create customized registries by automatically generating and deploying registries on blockchain. For users, there is a web forms-based interface, and a model that is not closely bound to the underlying blockchain technology. Regerator includes (1) a smart contract generator that can generate and deploy smart contracts representing registries on the Ethereum blockchain and (2) a generator for web-based RESTful APIs and user interfaces to interact with the generated registries. The feasibility of the approach is illustrated through a case study of applying it to an open data registry, using metadata from data.gov.au, and a registry model derived from an existing metadata registry platform.

8.3.1 Registries on Blockchain

Registries are authoritative databases for specific entities and are used to manage many aspects of daily life, such as land titles, business names, books, marriages, births and deaths, music, films, and domain names. Being an authoritative database means that a registry contains the default version of the truth. Sometimes a registry will be the legally authoritative source of truth, such as for land under a Torrens Title system.

Many public registries are hosted and maintained by government agencies whose authority guarantees authenticity for the registered entities. Every change to a registry is recorded with a digital fingerprint, which can be verified independently. A registry should store a history of all changes and be open to independent scrutiny. A registry may reference other registries to reduce duplication and errors. Registries should be highly available, because other registries and services depend on them. Open registries are publicly available, which means that the registry may be accessed, copied, or derived freely by the public. For instance, a business name registry, such as the Australian Business Register,[5] is a public registry whose entities can be requested by anyone at any given time. Building registries on blockchain can leverage key properties provided by blockchain and utilize the infrastructure of blockchain to achieve interoperability.

The main non-functional properties for registries on blockchain are as follows:

- **Integrity** concerns the accuracy and consistency of data over its entire life cycle. Data integrity is a key requirement of a registry, which means that the items can be only registered and changed by the authorized users. Many blockchain

[5] https://abr.gov.au/.

techniques are censorship-resistant, which helps to ensure the ongoing integrity of the full log behind the registry.

- **Availability** is also a key requirement for registries, especially national public registries, which form the basis for many other services that utilize the data from the registries. A blockchain system maintains consensus on data that is replicated across the network with many processing nodes. Therefore, there is no single point of failure since the infrastructure is fully decentralized.
- **Interoperability** is needed for registries to refer to and interact with each other and can be supported on a blockchain as it provides a common underlying infrastructure.
- **Efficient reading** is required to allow large-scale users of the registry to access local copies of the registry directly, to control latency and cost. On a blockchain, this is achieved because every node within the blockchain network has a local copy of all historical data. However, light users might find the cost of operating a full node relatively high, e.g. when compared to API calls.
- **Programmability** is required to allow more sophisticated, flexible, and finer-grained access control models to register and manipulate the items in the registry. On a blockchain this can be supported by using smart contracts. The computational results are verified by the participants of the network and recorded on blockchain, providing a full audit log of function calls in transactions as well as logical states of the registries.
- **Immutability** is required to enable an audit trail of all historical operations on the registry, to create complete traceability of records. This is a key property of blockchains. However, some registries need to provide functionality to remove records from the registry as if those records were never created, e.g. to respond to a court order for the removal of those records. This can be a challenge on a blockchain.

8.3.2 A Tool for Registry Generation: Regerator

In the remainder of this section, we discuss a tool called *Regerator*, which was developed in our research. Regerator is a model-driven framework for the generation of registries on a blockchain and for the generation of interface components for those registries. Currently it generates registries in Solidity for Ethereum. As a model-driven framework, it could support additional backend blockchain platforms in the future, provided that those platforms have sufficiently expressive smart contract languages. Regerator has three core components: a *smart contract generator*, a *registry of registries*, and interfaces for *smart contract management*, as shown in Fig. 8.5.

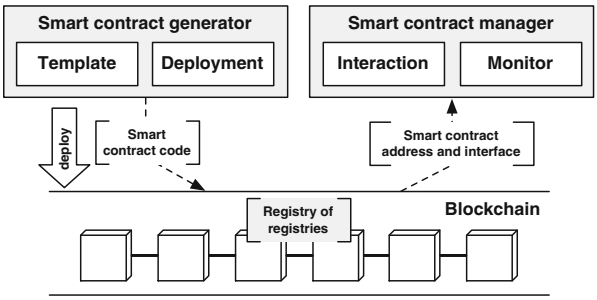


Fig. 8.5 Overview of registry generator on blockchain. © 2017 by the Commonwealth Scientific and Industrial Research Organisation, reprinted with permission

Smart Contract Generator

The smart contract generator allows the users of Regerator to generate smart contract registries from registry models and to deploy the generated smart contracts onto the blockchain. The smart contract model has four parts, including *basic information*, *registry type*, *basic operations*, and *advanced operations*.

- **Basic information** includes the registry name, description, and user-defined data fields and their types.
- **Registry type** can be 'single' or 'distributed'. The 'single' registry type holds all records as values in the data store for a singleton smart contract for the registry. The 'distributed' type manages each record as a separate smart contract. A main registry smart contract creates these contracts and stores pointers to them. The 'single' option is suitable for simple registries, while the 'distributed' option is suitable for registries with complex operations, such as finer-grained permission management at individual record level.
- **Basic operations** are the operations that can be performed on an individual record, including Create/Read/Update/Delete and existence checking. Users can configure whether or not a record is updatable. The Delete operation is a logical delete only, since it is impossible to remove historic records on a blockchain.
- **Advanced operations** include *access control*, *foreign key*, *version control*, *provenance*, *trading*, and *multi-signature*. We explain them in more detail below.
  - **Access control** is required to restrict users to certain operations. In the case of a public registry, only authorized government agencies are allowed to insert or update records, even though the registry is readable by the public. To enable permission management, a whitelist or a blacklist of addresses can be provided for the invocation of operations. We allow for the definition of access control mechanisms at the registry layer or the record layer. (For more restrictive read access control, a private blockchain can be hidden behind a web interface that implements those access control mechanisms, but this is not discussed here.)

We provide two types of access control management. The basic type is to check the permissions directly before executing an operation. The second type is to use a separate indirectly invoked *permission smart contract* as a gateway to manage a whitelist or blacklist; the operations of the registry then only check against the address of the permission contract. Deciding between these two alternatives depends on several factors, such as coupling, modifiability, and the size of the smart contract, which impacts the cost of deployment.

- **Foreign key** is a concept borrowed from relational database, which allows users to include the identity of a record from one registry as an attribute of a record to another registry as a way to define the relationship between two registries.
- **Version control** allows users to explicitly add a version number to an update on a registry and enables more efficient querying.
- **Provenance** in the context of registry refers to a log of all the operations that have been executed on a given registered entity. Such information is necessary for auditing data integrity. Blockchain-based registries naturally support provenance, as all data on the blockchain is immutable and valid.
- **Trading** or **transferring ownership** is required by registries that allow the trade of registered items, such as domain names registered in Domain Name System (DNS). This function is implemented as an escrow, which holds the money from the buyer first when they make an offer; when the current owner accepts the offer, the smart contract transfers the money to the seller and changes the ownership of the item to the buyer.
- **Multi-signature** requires multiple parties to jointly sign a transaction to invoke a smart contract operation. For instance, a publication registry like arXiv.org might require the permissions from all the authors of an article to update or delete the record. This function is planned for future work.

After registries have been defined, the smart contract generator provides a view to show the registries and the relationships among them as a graphical model. A screenshot of this functionality is shown in Fig. 8.6. The user can then decide to deploy the registries on blockchain.

Registry of Registries on Blockchain

The registry of registries stores references to all registries generated using Regerator on-chain. This facilitates version control of the generated registries. If a registered registry is updated to a new version, the address of the new smart contract is added to the registry of registries. Other tools and users can query the registry of registries to retrieve the current location and status of a registry or to view a historical version.

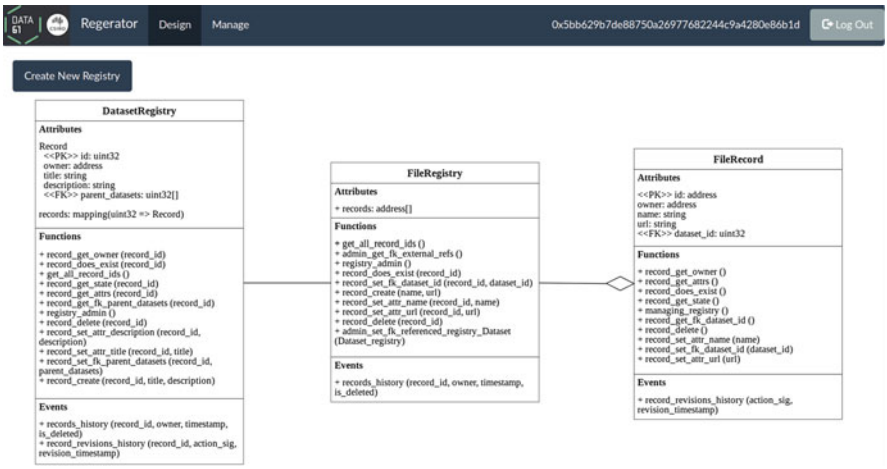


Fig. 8.6 Screenshot of the data model view of Regerator. © 2017 by the Commonwealth Scientific and Industrial Research Organisation, reprinted with permission

Smart Contract Manager

The smart contract manager provides web-based RESTful APIs and user interfaces to allow users to manage and interact with the generated registries. Similar to the business process execution approach discussed in the previous section, there is a dry-run mechanism that validates and tests the transaction for each of the functions defined in a registry by invoking the function on the local blockchain node behind the interface. If the output of the dry-run matches the user's expectation, the transaction is submitted into the blockchain network. This dry-run mechanism allows users to check the effect of their transactions before making permanent changes and incurring actual cost for submitting the transactions to the blockchain network. A smart contract monitor provides functionality to monitor contract events. In Ethereum, smart contracts can emit events and write logs to the blockchain when a transaction is processed. Tools and users can watch for new events, which show up on the page when there are events being recorded on blockchain during the contract execution.

8.3.3 Exemplar Case Study: Open Data Registry

To demonstrate the feasibility of the Regerator approach for model-driven generation of blockchain-based registries, we used Regerator to build a metadata

registry inspired by the Comprehensive Knowledge Archive Network (CKAN).[6] We populated this example registry with metadata taken from data.gov.au. We discuss some design considerations from the implementation as well as transaction cost below.

CKAN

CKAN is a web-based open-source data registration system, which provides functionalities to streamline publishing, sharing, finding, and using data. CKAN has been used by public institutions and governments to open their data to the general public, e.g. data.gov.au and data.gov.uk.

The central entity type in CKAN is a *package*. A *package* defines a variety of metadata of datasets, such as name, description, license, and tags. CKAN also supports an unlimited amount of customized metadata in the form of key/value pairs. The relationships between packages can be defined, such as *depends on*, *child of*, and *derived from*. Another entity type in CKAN is *resource*, which represents the raw data in the dataset, such as files or APIs. A *package* can be associated with multiple *resources*.

Implementation

We modelled elements of CKAN's metadata schema using Regerator and generated a blockchain-based registry system for the metadata of datasets. One architectural decision to be made is either to manage one entity as part of the attributes of another entity or to model both entities as separate registries. For the first choice, the nested entity will not have a unique, identifiable ID. As for the second choice, *foreign key* references between them need to be defined in order to encode the relationship, and both the entities can be uniquely identified. For the entity to be modelled as registry, another architectural decision to be made is either to model the entity as a 'single' registry or a 'distributed' registry. The factors to consider include the complexity of the data structure, the nature of the relationship between entities (coupling), and the cost of deploying and executing the registries on blockchain.

In the case of CKAN, there are potentially three entities that could be implemented as separate registries, including *package*, *resource*, and *organization*. Although *resources* are associated with a *package*, a *resource* is also an independent entity with its own metadata and can be managed separately. Thus, we have decided to record *resources* in a separate registry. Finally, *organization* is implemented as a separate registry that groups the address of all the users from the same organization. The organization registry can be used to define access, akin to role-based access control (RBAC).

[6] http://ckan.org/.

Table 8.1 Cost of using blockchain

	Registry deployment				Record creation (average)			
	Gas cost		Cost in US$		Gas cost		Cost in US$	
Entity	Single	Distr	Single	Distr	Single	Distr	Single	Distr
Organization	1.84M	2.54M	US$30.9	US$42.7	183k	0.93M	US$3.0	US$15.8
Package	1.84M	2.54M	US$30.9	US$42.7	340k	1.09M	US$5.7	US$18.5
Resource	1.78M	2.55M	US$29.9	US$42.7	302k	1.07M	US$5.0	US$17.8

Example Data

After implementing the blockchain-based registry, we queried the metadata of all the datasets from data.gov.au and added that to our registry to test the feasibility of our approach. Information about the number of each entity and the collected fields are shown as below.

- Organization (533 entries): *name*, *jurisdiction*, *spatial_coverage*, *email*, *telephone*, *website*
- Package (33,810 entries): *name*, *owner_org*, *license_id*, *contact_point*, *spatial_coverage*, *temporal_coverage*
- Resource (64,147 entries): *name*, *url*, *package_id*, *format*, *hash*, *size*

During the metadata import, we collected data about the blockchain cost as gas consumed (i.e. transaction execution cost) for deploying a registry and adding a record to the registry. We use this information to calculate the monetary cost of using blockchain as metadata repository according to the cost model of Ethereum. Table 8.1 reports the cost for the different design options ('single' or 'distributed' registry). The data also shows how different architectural decisions can affect the cost of deploying and executing the registry. We assume the gas price is 2×10^{-9} ETH (2 Gwei) and the exchange rate for Ether is US$420/ETH[7] as of 2 August 2018.

Discussion

Impact of Architecture Design on Cost On the Ethereum blockchain, the cost of creating a registry contract is comprised of fixed costs and variable costs. Fixed costs are the base amount for the transaction itself and the cost for allocating an address on the blockchain. Variable costs are affected by the architectural design of the registry contract, e.g. the cost of data payload. Similarly, the cost of adding records to a registry is also comprised of a fixed cost for the transaction itself and

[7]ETH is the currency code for Ethereum's cryptocurrency.

some variable costs including for the data payload and to execute the functions defined in the registry contract.

In contrast to existing practice, where adding a record is not normally independently accounted for financially, using a public blockchain means that adding a record costs real money (cryptocurrency). However, the blockchain ecosystem will retain this data indefinitely as long as the blockchain exists, at no additional cost. The most costly field (with the biggest size) of both *package* and *dataset* in our experiment was 'description', which amounted to approx. 85% of the total cost if included on blockchain. If it is not of high importance to store this information on-chain, storing it off-chain could significantly reduce the cost.

Interoperability In the ecosystem of CKAN, the datasets in different CKAN repositories refer to each other through importing the metadata from the referred repository to the primary repository and transferring it to the correct format due to the customer-defined fields. Regerator allows references to be defined as foreign keys, thus avoiding redundancy and preventing inconsistent drift.

8.3.4 Conclusion

In this section we discussed applying a model-driven approach for registries on blockchain. The Regerator system allows users to configure a registry model in a browser-based application and to automatically generate and deploy smart contract code implementing the registry on a blockchain. In addition, Regerator can also create user interfaces and RESTful APIs.

Execution cost for a generated registry is affected by architectural options represented within the registry model, and we have explored this through experiments on the Ethereum blockchain. The cost model for blockchains is different from conventional (cloud or in-house) servers, because transactions are expensive but data is retained indefinitely at no additional cost. Qualities like cost will be discussed in the next part of this book.

8.4 Summary

This chapter started with an argument about why model-driven engineering is particularly useful for blockchain-based applications: to avoid known vulnerabilities and technology lock-in, to implement best practices, and to facilitate understanding across parties and thereby increase trust in smart contract code.

We then elaborated on two methods for model-driven engineering: one for collaborative business processes and one for registries of non-fungible assets, like land titles or ownership of intellectual property. We also discussed how architectural

decisions impact qualities like cost and maintainability. The next part of the book will look at some of these qualities in more detail, starting with cost and how to estimate it.

8.5 Further Reading

For a brief summary of model-driven engineering, including its history and role in software engineering, see, e.g., Schmidt (2006).

Parts of this chapter are based on our own research publications, in particular the business process monitoring and execution approach (Weber et al. 2016), and the registry generator tool Regerator (Tran et al. 2017). For the business process part, we devised an optimized version of the approach (García-Bañuelos et al. 2017). This variant minimizes gas cost by trading it against lower readability of the code and lower isolation between process instances. It has been implemented in the tool *Caterpillar* (see López-Pintado et al. 2017). In contrast to these approaches, Prybila et al. (2017) present an approach to track flexible processes that can deviate from the model using the Bitcoin blockchain. Hull et al. (2016) propose to use an artefact-centric process modelling method for blockchain-based processes. Recently, the two model-driven engineering approaches discussed in this chapter have been combined in the Lorikeet tool (Tran et al. 2018).

Findings that the integration of business processes contributes to better operational and business performance are discussed in Flynn et al. (2010) and Narayanan et al. (2011). A lack of trust, however, may hamper collaborative process performance (Panayides and Lun 2009).

The supply chain scenario shown in Fig. 8.1 is derived from the literature (Fdhila et al. 2015).

The research literature on collaborative business processes has intensively investigated different notions of compatibility between the local processes of different partners and between local processes and a global process. Such compatibility can be achieved by design, for instance, using a P2P approach (van der Aalst and Weske 2001), transformations from a global choreography (Mendling and Hafner 2008; Weber et al. 2008), or interaction modelling (Decker and Weske 2011).

Business processes involve different trust issues (see, e.g. Viriyasitavat and Martin (2011) for a summary) which can be addressed in different ways. For example, Carminati et al. (2014) relaxed the assumption that the broker hosting the process engine has to be trusted: using selective encryption, data access for both the broker and the service partners can be restricted. Mont and Tomasi (2001) designed a trust service for cross-company collaboration based on a hybrid architecture mixing a trusted centralized control with untrusted peer-to-peer components. Li et al. (2010) put forward an agent-based architecture that can remove the scalability bottleneck of a centralized orchestration engine and provides more efficiencies by executing portions of processes close to the data they operate on. In virtual organizations,

Squicciarini et al. (2008) proposed to select partners on the basis of disclosure policies and credentials (i.e. identity attributes issued by a 'credential authority').

Key requirements and characteristics of registries were discussed in a UK Government report by Downey (2016). Regis[8] is a contract generator for registries on the Ethereum blockchain but only provides very basic operations.

[8] https://regis.nu/.

Part III
Quality Impact of Using Blockchain

Blockchain systems emerged to support financial transactions (digital currency), and so it is not surprising that the major supported non-functional properties (NFPs) are those that are critical in that domain: integrity and non-repudiation (including immutability of data, and transparency). As a highly distributed and redundant data store, blockchain systems can also support high levels of availability for reading data. As discussed earlier, there are some well-known limitations on NFPs for blockchain systems. Some are inherent to the technology, but others are only current limitations and may well be overcome in the near future. We discuss a variety of NFPs below.

Chapter 9
Cost

with Paul Rimba and An Binh Tran

In software architecture for blockchain-based applications, one of the most critical non-functional properties to consider is cost. The (monetary) costs of execution and storage are as important for blockchain technologies as they are for conventional technologies. However, blockchain systems have different kinds of cost models, and the cost for storing too much data on-chain can explode rather quickly. In this chapter, we discuss different options for storing data, and the principles of cost for smart contract deployment and execution.

Blockchains enable decentralized trust in storage and execution, but bring trade-offs against execution cost and latency. Therefore, we present mathematical cost models for blockchain and a particular cloud technology. Using these models, we investigate the question: *What kinds of cost trade-offs are there for blockchain vs. cloud?*

To illustrate answers to these questions below, we use an exemplar system of collaborative business process execution, implemented on both blockchain and cloud technologies. We use Ethereum because, like cloud platforms, it supports general purpose computation. In this chapter, we use the exchange rates of US$7650/BTC[1] and US$420/ETH[2] from August 2, 2018. We also assume a *gas price* of 2×10^{-9} ETH (2 Gwei) on Ethereum.

[1]BTC is the currency code for Bitcoin's cryptocurrency. Source for exchange rates is https://poloniex.com/exchange#usdt_btc.

[2]ETH is the currency code for Ethereum's cryptocurrency. Source for exchange rates is https://poloniex.com/exchange#usdt_eth.

X. Xu et al., *Architecture for Blockchain Applications*,
https://doi.org/10.1007/978-3-030-03035-3_9

9.1 On-Chain Data Cost

A common practice for data management in blockchain-based systems is to store raw data off-chain and to store on-chain just metadata, small critical data, and hashes of the raw data. We touched on this topic at various points of the book already, specifically in Section 6.3.3 and its comparison table (Table 6.2). Here we provide an in-depth discussion of the specific methods that can be used and the costs incurred.

In the Bitcoin blockchain, before *OP_RETURN*[3] was made a valid *opcode* (i.e. function of Bitcoin Script language) to store arbitrary bytes in an unspendable transaction, users were able to include limited information into transactions on-chain using one of four methods. These were: writing in a coinbase transaction which is only editable by miners, using the *nSequence* field, using a fake account address, or using unreachable script code defined through *if* and *else* conditions.

Four Ways to Store Arbitrary Bytes in an Unspendable Bitcoin Transaction
In the first method, every block has a coinbase transaction that mints new coins. The recipient of the coinbase transaction is the miner who generates the block. There is a parameter *coinbase* in the coinbase transaction, which can contain arbitrary data from the miner, and only the miner has access to this parameter.

In the second method, the blank field *nSequence* of a normal transaction is used to distinguish some transactions from other Bitcoin transactions, e.g. presenting assets other than the BTCs. Every participant which has the permission to submit transaction can set the value of *nSequence*.

For the third method, data can be encoded into a fake account address. The data is recorded on blockchain by sending a small amount of coins to the fake account. Any coin sent to the fake address is lost forever. One way to extend the mechanism is to use 1-of-n *multi-sig* transaction. Thus, if the recipient account of the transaction belongs to the owner of the arbitrary data, no coins are lost. To avoid denial-of-service attacks, Bitcoin sets a minimum amount of funds that can be transferred to an address, so that transactions with outputs below this threshold are discarded by the miners.

In the fourth method, smart contracts use conditional statements, such as in Bitcoin's Script or Ethereum's EVM. For example, Bitcoin Script has *OP_IF*, *OP_ELSE*, and *OP_ENDIF*. A clause within a conditional statement, which cannot be reached under any condition, can be used to store arbitrary data. This conditional statement causes extra overhead.

All four methods are deprecated now that *OP_RETURN* has been introduced as an official way to embed arbitrary data in a Bitcoin transaction.

Table 6.2 compares the *OP_RETURN* mechanism with other options provided by public Ethereum to store arbitrary data. There are trade-offs in cost efficiency, performance, and flexibility. The *OP_RETURN* instruction returns immediately with an error so that the included data is not interpreted as a script. The default Bitcoin client only relayed *OP_RETURN* transactions up to 80 bytes, which was

[3] https://bitcoinfoundation.org/core-development-update-5/.

reduced to 40 bytes in February 2014.[4] Storing 80 bytes of arbitrary data on the Bitcoin blockchain costs roughly US$0.459.[5] It is debatable whether Bitcoin should be used to record arbitrary data.

Ethereum, on the other hand, theoretically allows storing arbitrary structured data of any size. According to the cost model given in the Ethereum yellow paper, every transaction has a fixed cost of 21,000 gas (gas is the internal pricing for executing a transaction or storing data), and every non-zero byte of data costs additional 68 gas. Thus, the total cost of storing 80 bytes of data on Ethereum blockchain by submitting a transaction is 26,440 gas (assuming all bytes are non-zero), which is roughly US$0.22.

Ethereum provides two other ways to store arbitrary data in smart contracts. For 32 bytes of data, the first option is to store the data as a variable in a smart contract (all simple types in Solidity, the script language on Ethereum, are 32 bytes). The cost of storing data in the contract storage is based on the number of *SSTORE* operations required for the contract variable. In the case of storing 32 bytes, there is one *SSTORE* operation that changes the data from zero to non-zero, which costs 20,000 gas. As mentioned, the transaction as the carrier costs a base 21,000 gas. The data payload of the transaction including the function signature and the actual data costs extra gas. Other than these two costs, there is a cost for creating the smart contract depending on its complexity. In total, the cost is larger than US$0.036(20,000 + 21,000 + 32 × 68 gas). A subsequent transaction that updates the variable will incur 5000 gas, instead of 20,000 gas, for keeping the data as non-zero. Therefore, the subsequent transaction will be US$0.024 (5000 + 21,000 + 32 × 68 gas).

The second option is to store arbitrary data as a log event. This follows different rules for calculating cost. Logged data is stored in log topics which cost 375 gas, and where every byte of data in a log topic costs an extra 8 gas. Including the fixed cost of the carrier transaction with data payload, the rough cost of using a log event to store 32 bytes of data is US$0.018 (21,000 + 375 + 32 × 8 gas). Storing data as a variable in a smart contract is more efficient to manipulate but less flexible due to the constraints of the Solidity language on the value types and length. The flexibility and performance of using smart contract log events is intermediate because log events allow up to three parameters to be queried.

Finally, we reiterate that data storage on blockchain follows a different cost model than conventional data storage. Although it may seem more expensive, storing data on blockchain is a one-time cost for permanent storage. (However, note that Ethereum allows a partial refund on reclaimed smart contract variable storage.)

Selection of off-chain data storage concerns the interaction between the blockchain and the conventional data storage facilities. Off-chain data storage can

[4] https://github.com/bitcoin/bitcoin/pull/3737.

[5] Assuming a typical Bitcoin transaction with one input and one output, which has about 220 bytes, the default transaction fee rate of 2×10^{-4} BTC/KB (see https://en.bitcoin.it/wiki/Transaction_fees).

be through conventional enterprise IT systems, a private cloud on the client's infrastructure, or a public storage provided by a third-party. The flexibility of using cloud to store data depends on the implementation. Some peer-to-peer data storage facilities are designed to be friendly to blockchain, such as IPFS[6] and Storj.[7] IPFS is free, but ensuring availability requires providing an IPFS server that hosts the data. The cost of Storj is US$0.015/GB/month. In a peer-to-peer data storage, the data is replicated automatically by the peer-to-peer network, or based on the behaviour of users, e.g. data is replicated once a user accesses it. In a cloud environment, data replication needs to be managed by the system or consumer.

9.2 Smart Contract Cost

There is a cost charged on Ethereum for transactions in relation to their complexity. A detailed cost model is presented in Section 9.3.1. In rough terms, there is a fixed base cost for any transaction, the 21,000 gas mentioned above, plus variable components: data attachments as discussed above; executing a smart contract method is charged per bytecode instruction; and additional cost arises during deployment of new contracts. All costs in Ethereum follow a fixed pricing table, specified in the unit *gas*. Gas cost is converted to Ether, Ethereum's own cryptocurrency, with a user-defined *gas price* factor, i.e. how much Ether-per-gas the creator of the transaction is willing to pay. By default, Ethereum clients set the gas price to the current market rate, an average over previously included transactions.

To prevent denial-of-service attacks, Ethereum has a *block gas limit*: the sum of gas used by the set of transactions included in a given block cannot exceed this limit. The block gas limit is set by the miners. Each miner winning a block can slightly increase or decrease the block gas limit or keep it unchanged. Because the block gas limit is defined in terms of gas usage, not the transaction fee in Ether, *this limit cannot be influenced by variations that the user has power over* (such as underbidding the market price), effectively making it a limit of complexity for new blocks. As such, the block gas limit acts also as an *upper bound to throughput scalability*. But since the cost of transactions can vary, it is non-trivial to understand how that bound relates to transaction throughput for a given application.

9.3 Cost Models

In this section, we describe models (formulae) to estimate the cost of running an application on two different types of infrastructure. We use the execution of an instance of a business process model as sample application. For blockchain

[6] https://ipfs.io/.
[7] https://storj.io/.

infrastructure, we use Ethereum because its smart contracts are in a Turing complete programming language which can be used to represent business process logic—see Chapter 8. For conventional cloud infrastructure, we use Simple Workflow Service (SWF) from Amazon Web Services (AWS), because it is dedicated to process execution and offered by a leading commercial cloud computing provider (i.e. Amazon). It can implement the commonly used workflow patterns, as well as synchronous and asynchronous messaging patterns.

Executing collaborative processes across organizations requires three types of components: (i) the implementation of the collaborative process model, to coordinate the work across participants; (ii) implementations of the activities that participants perform; and (iii) interfaces (or triggers as in Section 8.2) that control interactions between the collaborative process and the participants' activities. The details of a participant's activities (ii) are typically shielded from external organizations and given a good interface (iii) are independent of the choice of coordination technology. As such, we disregard factor (ii) in our cost models.

First, consider (i), the cost overhead of process coordination. For a cost model for a single instance of a coordinating process, both for blockchain and Amazon SWF, we send all messages synchronously and in a way that conforms to the business process model. Next, consider (iii), the cost for running a virtual machine (VM) that hosts the interface between the coordinating process and the internal systems. This is dependent on the choice of technology as well as on the workload. When the workload exceeds the VM's capacity, a more powerful or additional VM will be required. Blockchain infrastructure also needs a 'full node' of the blockchain, which is relatively heavyweight.

We describe the two components of the cost model, first for Ethereum in Section 9.3.1 and then for SWF in Section 9.3.2.

9.3.1 Ethereum Blockchain Cost Model

There are three types of transactions in Ethereum: financial transfer, message call, and contract creation. Each has the following basic elements: `from`, `to`, `gasLimit`, `value`, and `data`.

The `from` and `to` fields signify the sender and the recipient of the transaction, respectively. For a *financial transfer* transaction, the amount transferred is given in the `value` field. The `data` field is optional but can contain data in arbitrary other forms, e.g. XML, pictures, or MP3s. The fee for a transaction with attached data covers the cost for storing the data permanently in the blockchain and is proportional to the size of the data—see the details in Section 9.1. A *message call* transaction invokes a function of a contract, where the `data` field carries the method to be invoked and the parameters. The `gasLimit` is used to specify the maximum gas that can be used in this transaction. *Gas* is paid for each bytecode instruction that is executed. Finally, a *contract creation* transaction is indicated by a `to` value of

NULL and `data` that contains the contract bytecode. For both *message call* and *contract creation* transactions, the `value` field is optional.

We divide our business process blockchain cost model into two parts, one for the cost of deploying a smart contract and one for the cost of executing business process coordination.

A contract creation transaction includes compiled bytecode in the `data` field, and the permanent storage of this data incurs cost. An optional 'endowment' can be provided, so that the new contract has a positive balance upon initialization. When a contract is created, a particular Ethereum address is assigned to it, which is subsequently used to interact with that contract. This contract address is calculated with a deterministic function that depends only on the creator's Ethereum account.

The details of the costs of contract creation are outlined in the Ethereum yellow paper. We refer to this cost as $\mathcal{C}_{create}$. A contract creation transaction costs a base amount of 21,000 gas for the transaction itself ($\mathcal{C}_{tx}$), plus 32,000 for allocating a new address ($\mathcal{C}_{addr}$), plus the cost of data payload ($\mathcal{C}_{pload}$, the size of contract bytecode multiplied by gas per byte), plus any additional gas that is consumed by the opcodes in the function definition ($\mathcal{C}_{fn_{def}}$). The contract creation cost formula is shown in Eq. (9.2). An online tool is provided by Ethereum to estimate the amount of gas required in Ethereum. At the time of writing, the cost of payload for contract bytecode is 200 gas per byte, while the cost of payload for data in a financial transaction and message call is 68 gas per non-zero byte and 4 per zero byte.

$$\mathcal{C}_{pload} = payload\,(in\ bytes) \times \mathcal{C}_{gas/byte} \tag{9.1}$$

$$\mathcal{C}_{create} = \mathcal{C}_{tx} + \mathcal{C}_{addr} + \mathcal{C}_{pload} + \mathcal{C}_{fn_{def}} \tag{9.2}$$

In Ethereum, a contract can create another contract. This is cheaper because this does not incur $\mathcal{C}_{tx}$. So, the cost of creating a new contract by an existing contact, $\mathcal{C}_{create_{internal}}$, can be calculated as shown in Eq. (9.3).

$$\mathcal{C}_{create_{internal}} = \mathcal{C}_{addr} + \mathcal{C}_{pload} + \mathcal{C}_{fn_{def}} \tag{9.3}$$

The second part of our cost model concerns the cost for executing the coordinating business process and is summarized in Eq. (9.4). A coordination message is treated as a function call in Ethereum. A function call costs a base amount of 21,000 gas for the call itself, plus any additional gas that is consumed by the opcodes present during the function execution ($\mathcal{C}_{fn_{exec}}$) and the cost for the data payload.

$$\mathcal{C}_{coord} = \mathcal{C}_{tx} + \mathcal{C}_{pload} + \mathcal{C}_{fn_{exec}} \tag{9.4}$$

The costs calculated with Eqs. (9.1)–(9.4) are in gas. In order to convert these costs into *Ether*, the digital currency of Ethereum, the total gas consumed must be multiplied by the gas price in *wei* (one wei is 10^{-18} Ether). Finally, the cost in

Ether can be converted into another currency through an exchange service at some exchange rate, $EXC_{ETH2CUR}$. We specify this in Eq. (9.5).

$$\mathcal{C}_{in\$} = \mathcal{C}_{inGas} \times gasPrice \times 10^{-18} \times EXC_{ETH2CUR} \tag{9.5}$$

Equations (9.2) and (9.4) are concerned with the setup and coordination cost for component (i) in the introduction of this section, for blockchain infrastructure. Component (ii) is disregarded as explained, but component (iii) needs to be considered: the cost of the VM that acts as an interface between the process and the enterprise systems of participants.

To calculate the cost of the interface VM, we need a few more definitions. Let $EC2_t$ be the set of all available VM types in AWS Elastic Compute Cloud (EC2)[8] and $ec2_t \in EC2_t$. We define each VM type's capacity as $\mathcal{TP}_{bc} : EC2_t \mapsto \mathbb{R}$. Next, we define a function that determines the VM type based on the coordination workload, WL_{bc}, and VM capacity: $f_{bc} : (\mathcal{TP}_{bc}, WL_{bc}) \mapsto EC2_t$. The cost of running a VM of this type per billing time unit (BTU) is captured as $EC2_{price} : EC2_t \mapsto \mathbb{R}$. Finally, we obtain the VM cost by multiplying the price with the number of BTUs it is required to run, as shown in Eq. (9.6).

$$\mathcal{C}_{comp} = EC2_{price}(ec2_t) \times time \tag{9.6}$$

Note that, in the blockchain setup, the interface VM operates a full node. As such, if the VM is not constantly online, the required duration for this VM needs to include the time to synchronize the blockchain with the network. Ethereum clients have a 'fast' flag that allows faster synchronization: instead of downloading the full set of known blocks, only transaction receipts from blocks are downloaded. The receipts show that these transactions happened but do not show the results of the smart contract function executions, so provide less evidence for integrity. This can only be done when downloading the blockchain from scratch and takes on the order of hours to days for the public Ethereum blockchain, depending on the machine and connectivity chosen and the size of the data structure.

9.3.2 Amazon SWF Cost Model

AWS provides a service for workflow execution, called Simple Workflow Service (SWF), which we use as a representative for cloud-based business process execution. We chose SWF as it provides a clear mapping to our process model, for comparison. SWF has a tiered pricing model,[9] i.e. more usage will result in cheaper

[8] https://aws.amazon.com/ec2/instance-types/; AWS calls VMs 'instances'. To avoid confusion with process instances, we use the term 'VMs' instead.

[9] https://aws.amazon.com/swf/pricing/.

cost per unit. It has the following main elements: workflow, actor, task, and signal. A workflow is a collection of activities that can be performed by different actors in a specified sequence. A workflow in SWF represents an instance of a business process, while actors play participant roles from the business process. There are two different types of task: activity and decision. An activity task is used to schedule a notification to the appropriate actors to proceed with the next activity in the workflow execution. A decision task is used to determine whether the current state of execution conforms to the workflow and to determine which activity to execute next. A signal is an externally triggered event to a currently executing workflow. Table 9.1 shows the mapping of a business process to elements of Blockchain and SWF.

Figure 9.1 is a sequence diagram that shows how an example of a supply chain business process is executed using Amazon SWF workflow. Every actor involved in the business process implements its own trigger, which is a program that interacts with Amazon SWF through AWS API calls. When a trigger's API is called by its owner, the trigger translates the message into an Amazon SWF signal. SWF then schedules a decision task to evaluate the signal's content and to perform conformance checking. If successful, an activity task is scheduled to notify the actor of the next business activity. Task execution requires the actor to have a running *Amazon SWF worker* module, which can be operated either on AWS EC2

Table 9.1 Business process mapping to Amazon SWF elements and blockchain elements

Business process	Blockchain	Amazon SWF
Process instance	Instance of smart contract	Workflow
Conformance checking	Contract execution (partial)	Decision task
Activity	Contract execution (partial)	Activity task
Incoming message	Transaction	Signal
Outgoing message	Entry in contract event log	Notification

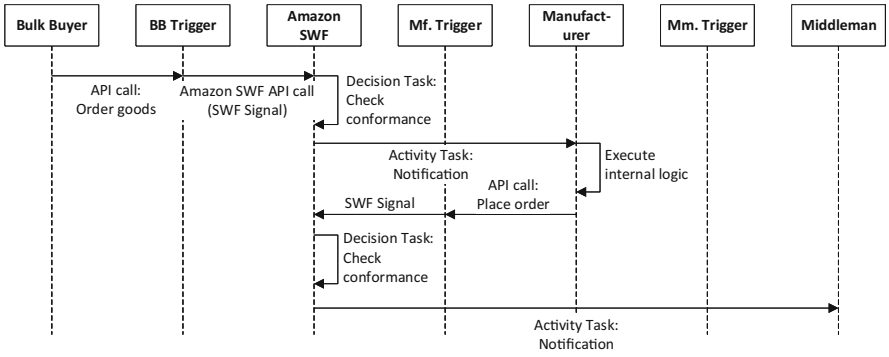


Fig. 9.1 Supply chain process implemented using Amazon SWF Workflow (cf. Fig. 8.4). © 2018 by Springer International Publishing, part of Springer Nature, reprinted with permission

or the actor's own infrastructure. This *Amazon SWF worker* module will execute both decision task and activity task scheduled by SWF. Conformance checking is a technique in process mining, which compares an existing process model with an event log produced by the process model. Conformance checking is used to check if what happened in reality conforms to the process model, and can be used at runtime.

The total cost for SWF-based execution has several components. First, the base cost for workflow instances $\mathcal{C}_{wf}$ can be calculated by multiplying the number of instances with the SWF cost of starting a workflow execution ($\mathcal{SWF}_{wf}$) as shown in Eq. (9.7).

$$\mathcal{C}_{wf} = \#wf \times \mathcal{SWF}_{wf} \tag{9.7}$$

The *execution* of activity tasks is done by the SWF worker, which we discuss below. The cost for *scheduling* tasks, $\mathcal{C}_{task}$, is the price per task ($\mathcal{SWF}_{task}$) multiplied with the sum of activity tasks and decision tasks that are executed; see Eq. (9.8). Note that the number of activities in a process instance equals the number of SWF activity tasks, whereas the number of decision tasks is that number *plus one* additional decision task (immediately after the start of the workflow instance).

$$\mathcal{C}_{task} = (\#actTask + \#decTask) \times \mathcal{SWF}_{task} \tag{9.8}$$

The number of signals can be obtained from the number of activities in a business process instance. We can calculate the cost of signals, $\mathcal{C}_{sig}$, by multiplying the number of signals with the price per signal, as shown in Eq. (9.9).

$$\mathcal{C}_{sig} = \#signals \times \mathcal{SWF}_{signal} \tag{9.9}$$

Data generated during the workflow execution is retained by SWF for a user-specified duration after completion of workflow execution (*retT*) and is charged for storage per 24 h. The workflow execution time (*execT*) is also charged per 24 h at the same rate as data retention cost ($\mathcal{SWF}_{ret}$). This is reflected in Eq. (9.10). Finally, cost of data transferred, $\mathcal{C}_{dat}$, inwards and outwards during the workflow execution, is the total payload data size (*payload*) multiplied with the cost per data unit ($\mathcal{SWF}_{data}$). See Eq. (9.11).

$$\mathcal{C}_{ret} = (execT + retT) \times \mathcal{SWF}_{ret} \tag{9.10}$$

$$\mathcal{C}_{dat} = payload \times \mathcal{SWF}_{data} \tag{9.11}$$

The formula to calculate the total cost of business process execution on Amazon SWF is shown in Eq. (9.12), which is the sum of individual costs incurred from Eqs. (9.7) to (9.11).

$$\mathcal{C}_{swf} = \mathcal{C}_{wf} + \mathcal{C}_{task} + \mathcal{C}_{sig} + \mathcal{C}_{ret} + \mathcal{C}_{dat} \tag{9.12}$$

Equation (9.12) provides the coordination cost when using SWF services and does not include the cost of the VMs to run the triggers and the Amazon SWF workers. In order to calculate the cost for the VMs, we first need to determine the VM type required for a specific workload, WL_{swf}. For that we again define the throughput per VM type as $\mathcal{TP}_{swf} : EC2_t \mapsto \mathbb{R}$. The throughput values here are different from the ones for blockchain triggers, due to the different modules that are running for SWF. Analogous to the blockchain calculations, we determine the required VM type based on the capacity of VM types and the workload: $f_{swf} : (\mathcal{TP}_{swf}, WL_{swf}) \mapsto EC2_t$.

We use that information to calculate the cost for running the VMs for the time needed as per Eq. (9.6). The minimum requirement is one VM to host the trigger and worker, with the caveat that all participants trust this VM. In a preferable setup, each participant involved provisions at least one VM to host their own trigger and worker.

In the cost model, we need to know the maximum throughput of each different VM type, $\mathcal{TP}$, for process execution on both blockchain and Amazon SWF. These need to be established through benchmark tests.

9.4 Using and Evaluating the Cost Model

In this section, we show how the cost models can be used to compare costs. We use the example of business process execution on Ethereum and Amazon SWF. We also describe some benchmark experiments that allow us to explore the accuracy and limitations of the cost models. Finally, we discuss how we can use the models to conduct sensitivity analyses, to better explore the cost consequences of the design in different business scenarios.

9.4.1 Experiment Setup, Methodology, and Benchmarking

For the cost comparison experiment, we use two datasets. The first is a process for incident management from the literature. Figure 9.2 shows the business process model, which has nine tasks and six gateways. The model has four conforming traces, all of which we use. Such a process would be cross organizational if, e.g. first-level support was outsourced. The second dataset is based on a real-world invoicing process, provided to us by the Minit process mining platform in the form of a log file with 5316 traces that comprise 65,896 events. We derived a process model from these with standard process discovery methods. The model has 40 tasks and 18 gateways. Due to the presence of loops, there is an infinite number of conforming traces. We use that model and the 5316 traces.

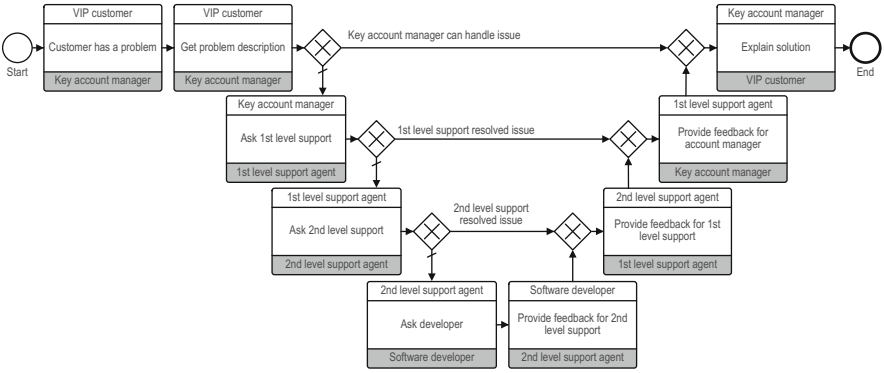


Fig. 9.2 Incident management case study workflow, adapted from literature (Notation: BPMN). © 2017 IEEE. Reprinted, with permission, from Rimba et al. (2017)

The test instances for the business process are read from a message trace log file. For each log line, we send a message to the respective actor's business process trigger, which sends a transaction to the blockchain or a signal to Amazon SWF.

Blockchain

For the incident management process, we reuse the results from our previous work, with experiments on the public Ethereum blockchain. For the invoicing process, we ran separate large-scale experiments. In both scenarios, each actor maintains a local Ethereum node, running go-Ethereum (geth). For incident management/invoicing respectively, we used geth versions 1.3.5/1.5.4, connected to the public/a private Ethereum blockchain, and compiled our smart contracts using Solidity compiler version 0.2.0/v0.2.1 with optimization enabled. We implemented the triggers in Node.js using the Ethereum library web3 version 0.15.1 for both processes.

Amazon SWF

For Amazon SWF, each actor was implemented with a business process trigger in Java, using the AWS SDK for Java version 1.11.13. This trigger calls the Amazon SWF API to send signals to the Amazon SWF. We deployed the trigger and SWF worker on an EC2 t2.micro VM.

VM Throughput Measurements

As mentioned in the previous section, we need to establish the maximum throughput of each different VM type, $\mathcal{TP}$, in order to use the cost model. We have run

Table 9.2 AWS EC2 VM types and specification

VM types	vCPU specifications	Memory (GiB)
t2.small	1 Intel Xeon E5-2676 2.40 GHz v3 w/ Turbo up to 3.3 GHz	2
m3.medium	1 Intel Xeon E5-2670 2.50 GHz v2 (Ivy Bridge) processors	3.75
m3.large	2 Intel Xeon E5-2670 2.50 GHz v2 (Ivy Bridge) processors	7.5
m3.xlarge	4 Intel Xeon E5-2670 2.60 GHz v2 (Ivy Bridge) processors	15

Table 9.3 Throughput experiment result

	Blockchain			Amazon SWF	
Metrics	t2.small	m3.medium	m3.large	m3.medium (default)	m3.medium (incr. limit)
Transactions or signals	13,580	7336	20,104	73,871	152,404
Network in (MB)	102	114	128	138	168
Network out (MB)	195	131	278	353	376
Duration (s)	3610	3605	3604	3605	3605
Average Tx/s or Average signal/s	3.8	2.0	5.6	20	42

benchmark tests to empirically determine this, using synthetic load based on the incident management process described earlier. We do not present the details of this test here but do show the results below. We used several types of EC2 VMs for a private Ethereum deployment. We used t2 VMs and m3 VMs which provide a consistent baseline performance for general purpose applications. All the VMs used solid state drives as disks. The specifications for the VM types are shown in Table 9.2. Table 9.3 summarizes the results for both blockchain and Amazon SWF.

9.4.2 Blockchain Results

For the invoicing process, we deployed a factory contract and ran 5316 process instances with a total of 65,896 transactions. As per our business process execution approach, when invoked for process instantiation, the factory contract generates a new instance smart contract, which contains the blueprint of the business logic. This smart contract also performs conformance checking during execution: for each transaction after instantiation, the process instance contract checks if this transaction is expected in the current state of the instance. There are 49 unique traces, i.e. 49 different paths through the process model were explored during the experiment. The deployment of the factory contract costs 0.0031 Ether (approx. US$1.30), and each unique trace has different costs associated to it, ranging from 0.0006 to 0.0017 Ether. We ran this experiment on a private blockchain where it cost a total of 15.66 Ether (approx. US$ 6577.20 on public Ethereum). Our private blockchain uses the same code as the public one, including for cost calculation.

For the incident management process, we refer to experiment runs that were reported previously (Weber et al. 2016). In these experiments on the public Ethereum blockchain, we ran 32 process instances with a total of 256 transactions. With a gas price of 20 Gwei, which was the market rate at the time (March 2016), the deployment of the factory contract costs 0.032 Ether, and each run of the incident management process, with data transformations, cost on average 0.0347 Ether. At the time when we conducted the experiment, the exchange rate was around US$10/ETH, and these costs equated to about US$0.30–0.40. With a current gas price of 2 Gwei and exchange rate of US$420ETH, the costs are approx. US$1.34 and US$1.46, respectively. The sharply increased exchange rate has, to a degree, been compensated by a lower market gas price.

9.4.3 Amazon SWF Results

In the SWF experiment, we created a new process instance (SWF workflow instance) for each run. On receiving a signal, Amazon SWF schedules a *decision task* for the worker. The worker checks the received signal for conformance with the business process implemented in the workflow and the state of the instance, and if successful progresses the workflow state accordingly.

If the signal frequency during the execution of a workflow instance is too high, AWS may schedule the next decision task to handle the decision logic for all the received messages in a batch. SWF would thus allocate a single decision task to handle multiple signals for a single workflow instance at once, which could distort our results. To prevent SWF from batch processing the signals for a single workflow instance, we send messages synchronously: once the result has been received, we send the next message for that instance.

We deployed an EC2 t2.micro VM for the trigger and the Amazon SWF task worker and executed process instances in sequence. For each process instance, the initialization creates a new workflow (instance) and a decision task to instruct the workflow to wait for the first signal. For each additional message, the trigger sends one signal which results in one activity task and two decision tasks: the workflow schedules a decision task each time it receives a signal or a completion message from an activity task. Thus, for X process instances with a total of Y events, there are X workflows, $Y - X$ signals and activity tasks, and $2Y - X$ decision tasks.

In our experiments, we set both the data retention rate and workflow execution to 1 day. The total cost for the invoicing experiment with 5316 process instances was US$7.23. This equates to an average cost of US$0.0014 per process instance, with 1-day data retention. The cost per process instance would be US$0.00318 if we increased the data retention to 365 days. Table 9.4 shows the cost breakdown. Amazon SWF data transfer is charged per GB, with 1 GB as the lowest denomination. As we incurred 4522 MB of data transfer during the Invoicing experiment, the cost for data transfer is rounded up to 5 GB.

Table 9.4 Amazon SWF cost breakdown—invoicing

Elements	Elements in experiment	Unit cost (US$)	Total cost (US$)
Decision task	126,476	0.000025	3.16
Activity task	60,580	0.000025	1.51
Signal	60,580	0.000025	1.51
Workflow	5316	0.0001	0.53
Retention (24 h)	5316	0.000005	0.027
Execution time (24 h)	5316	0.000005	0.027
Data transfer	5	0.09	0.45

Table 9.5 Amazon SWF cost breakdown—incident management

Elements	Elements in experiment	Unit cost (US$)	Total cost (US$)
Decision task	15,000	0.000025	0.375
Activity task	7000	0.000025	0.175
Signal	7000	0.000025	0.175
Workflow	1000	0.0001	0.1
Retention (24 h)	1000	0.000005	0.005
Execution time (24 h)	1000	0.000005	0.005
Data transfer	1	0.09	0.09

For the incident management process with 1000 process instances, the total cost for the experiment was US$0.925, resulting in an average cost of US$0.000925 per process instance. If we increased the data retention to 365 days, the cost per process instance would be US$0.002745. Table 9.5 shows the cost breakdown. The data transfer volume for incident management was 358 MB, which is rounded up to 1 GB.

9.4.4 Completeness, Correctness, and Comparative Analysis

We believe the cost model is complete because in our implementations of both variants we did not encounter any cost that is (1) not part of the cost models and (2) specific to either system. Take, for instance, broadband network access from the enterprise systems: we take that as a given, and there are no particular differences between the network requirements for blockchain or Amazon SWF.

In terms of correctness, we found the outputs of the 5348 process instances from both blockchain experiments to be consistent with the outputs of the Amazon SWF experiments, given the same inputs.

For the invoicing process, executing one process instance costs US$0.001359 on average in Amazon SWF. In comparison, executing the same process instance on Ethereum costs on average 0.00294 Ether, or approx. US$ 1.24, plus 0.0031 Ether (US$1.30) as a one-time cost for deploying the factory contract. Excluding the one-time factory contract deployment, the cost per process instance on blockchain is currently three orders of magnitude higher than on Amazon SWF. Blockchain stores the result in perpetuity (as long as the blockchain is in existence), while SWF has a 90-day limit on data retention. To put the higher *one-time cost for executing* a process instance on Ethereum into perspective with the *ongoing cost for data storage* on Amazon SWF: to reach break-even, the data would have to be stored for 243,863 days or approx. 668 years.

Similar findings are observed in the incident management process, where executing one process instance costs US$0.000925 on average on Amazon SWF. In comparison, executing the same process instance on Ethereum costs on average 0.00347 Ether, or approx. US$ 1.46, plus 0.0032 Ether (US$1.34) as a one-time cost for deploying the factory contract. The cost per process instance on blockchain in the incident management process is about three orders of magnitude higher than on Amazon SWF. The data needs to be retained for 266,447 days or approx. 730 years to reach break-even.

The Ethereum blockchain cost estimates from the online tool have a difference of up to $\pm 2.4\%$ for the contract creation part, i.e. factory contract and process instance deployment. For the cost of coordination ($\mathcal{C}_{coord}$), the online tool estimates this as transactional cost, which is the execution cost ($\mathcal{C}_{fn_{exec}}$) + 21,000 gas ($\mathcal{C}_{tx}$) + cost of payload ($\mathcal{C}_{pload}$). The cost of payload is (4 bytes of function signature + parameters in bytes) $\times$ $\mathcal{C}_{gas/byte}$. For most of the activities in incident management, our cost model can estimate the gas usage accurately, with the exception of *customer_has_problem* activity which has an unusual gas refund behaviour that affects the calculation of the execution cost by 15,000 gas. To achieve accurate gas usage and cost estimation for function execution, this is best achieved by deploying a private Ethereum blockchain. In a private blockchain setting, the conversion to fiat currency (Eq. (9.5)) may not be required.

The Amazon SWF cost model is accurate in estimating the costs for the SWF elements, with a possible variation for workflow execution time and data transfer. Estimating the cost of VM based on the maximum throughput of the VM type and the workload may vary due to performance variation in AWS EC2 and complexity of the activity task implementation.

One of the benefits of having a cost model is the ability to predict the cost for different workload settings. Having previously validated the cost models for Ethereum blockchain and Amazon SWF, this gives us all the components needed for us to predict the cost of business process execution for different workloads.

Table 9.6 Cost of blockchain experiments for invoicing and incident management processes under different exchange rates

Costs	Ethereum (in Ether)	Exchange rate (in US$)				
		0.10	1.00	10.00	100.00	1000.00
Incident management (contract deployment)	0.0032	0.00032	0.0032	0.032	0.320	3.20
Incident management (per process instance)	0.00347	0.000347	0.00347	0.0347	0.347	3.47
Invoicing process (contract deployment)	0.0031	0.00031	0.0031	0.031	0.31	3.10
Invoicing process (per process instance)	0.00294	0.000294	0.00294	0.0294	0.294	2.94

9.4.5 On the Volatility of Cryptocurrency to Fiat Currency Exchange Rate

The results of our comparative analysis are sensitive to the volatility of the exchange rate from cryptocurrency (Ether in our case) to fiat currency (US$ in our case). In order to illustrate this, consider a sensitivity analysis where we set the exchange rates for Ether to US$ in logarithmic scale from US$0.1 to US$1000. Another parameter we can vary is the retention rate, where we calculate for the cost of 24-h and 99 years (long-term) data retention. Table 9.6 shows the predicted costs of business process execution for both invoicing and incident management processes on Ethereum blockchain in this parameter space.

The costs of business process execution for both the invoicing and incident management processes on SWF with 24-h retention rate are US$0.001359 and US$0.000925, respectively. For 99 years retention rate, invoicing process will cost US$0.182029, and the incident management process will cost US$0.181595.

Blockchain and SWF costs are compared under different exchange rates (for blockchain) and different retention rates (for SWF) in Table 9.7. For the invoicing process, the cost on SWF (with long-term data retention) is two orders and one order of magnitude higher than Blockchain if the exchange rate is US$ 0.10 and US$1.0, respectively. This is consistent with our finding for the incident management process, where the cost on SWF is also two orders and one order of magnitude higher than Blockchain with the same exchange rates.

Table 9.7 Cost comparison of SWF and blockchain for invoicing and incident management processes under different exchange rates and retention rates

Costs	SWF cost (in US$)	SWF vs. blockchain cost comparison in ratio with different exchange rates (ratio < 1 means blockchain is cheaper)					Break-even rate (in US$)
		$0.10	$1.00	$10.00	$100.00	$1000.00	
Incident (24 h)	0.000925	0.375 (0)	3.751 (+1)	37.51(+2)	375.14(+3)	3751.35(+4)	0.27
Incident (99 years)	0.181595	0.002 (−3)	0.019 (−2)	0.191(−1)	1.91(0)	19.11(+1)	52.33
Invoicing (24 h)	0.001359	0.216 (−1)	2.16 (0)	21.63(+1)	216.34(+2)	2163.36(+3)	0.46
Invoicing (99 years)	0.182029	0.0016 (−3)	0.016 (-2)	0.162(−1)	1.615(0)	16.15(+1)	61.91

Positive numeric values in the brackets signify order of magnitude higher more expensive on blockchain.

9.5 Discussion

Below we discuss the cost of business process execution on Amazon SWF vs. Ethereum blockchain. Section 9.5.1 looks at why we might ever consider using blockchain if it costs orders of magnitude more than cloud services. We then look into trade-offs between cost and maintainability for different smart contract configurations on blockchain. Scalability of blockchain and SWF is discussed in Section 9.5.3. Finally, we discuss possible cost savings and improved throughput by improving the business process execution in Section 9.5.4.

9.5.1 Cost of Distrust

We have seen that blockchain costs orders of magnitude more than cloud services for realistic uses for business process execution. So why should anyone use a blockchain for this? The key difference is that blockchain technology can provide a trustworthy storage and execution environment, without requiring trust in any single third-party organization. In contrast, conventionally participants who do not yet know or trust each other need to jointly agree on a mutually trusted third-party. In the SWF setup, participants need to trust both AWS (for confidentiality and truthful execution) and the party controlling the Amazon SWF account in which the process is hosted.

This is of particular interest in situations of *coopetition*, where organizations cooperate for specific cases where achieving some business goals is mutually beneficial but compete in other cases. In our age of globalization, high market pressure, diversified organizations, and complex business networks, coopetition is a common situation. If multiple parties come together to achieve a joint goal, but some are in coopetition, it is important that the entity executing the joint business process is neutral.

Blockchain can be used to enable 'trustless' collaboration as it is not controlled by a single entity. However, as our experiments in Section 9.4.4 show, this comes at a premium price that can be three orders of magnitude higher than using cloud services like Amazon SWF. Although blockchain provides pseudonymity, companies involved in the coopetition will need to share their addresses (e.g. contract addresses and wallet) that will be involved in the business process. These addresses preserve their pseudonymity to other users of Ethereum.

Public blockchains inherently support payments and escrow handling. Due to a flat fee structure in blockchains, sending cryptocurrency along with existing messages *would not incur any additional cost*. This can offset the premium *cost of distrust* offered by blockchain. Commercial escrow services often charge 0.5–3.25%. Depending on exchange rates and amounts to put into escrow, blockchain's flat fees may actually *lower* the cost of process executions involving monetary transaction despite the additional cost of smart contract execution and data storage.

9.5.2 Cost vs. Maintainability

Different contract deployment methods impact cost and other non-functional properties. To illustrate this, we set out two sample configurations: (1) one smart contract with two functions and (2) two smaller contracts, each implementing one function where one of the contracts acts as an entry point. Both these configurations provide the same functionality, and intuitively the function definition costs should be the same. However, the first configuration will have lower deployment cost, even in terms of payload cost, than the second configuration. This is due to several reasons:

- For (2), one has to pay $\mathcal{C}_{tx}$ and $\mathcal{C}_{addr}$ twice.
- The total payload of the contracts in (2) is higher than in (1), as there are header bytes in the payload.

The trade-off is between cost and maintainability. (1) is cheaper but is not as maintainable as (2). If one of the functions needs to be modified, in (1) the updated contract needs to be redeployed as a whole. In contrast, in (2) only one of the contracts needs to be redeployed. Redeployment of a contract means getting a new address for the updated contract. In (1), the triggers need to be updated to point to the address of the updated contract, whereas in (2) this can be avoided.

9.5.3 Scaling Triggers for Blockchain and SWF

With increasing workload, additional resources are needed to accommodate the triggers' workloads. Two common ways to add resources are vertical (bigger VM) and horizontal scaling (more VMs). We discuss these for blockchain and SWF.

Blockchain nodes can scale vertically in order to accommodate increasing workload. However, horizontal scaling has complications. Although it is easy to add additional VMs into the network, using one account actively from multiple VMs may lead to what could be considered a double-spending attack. One way around this may be to use different accounts on different VMs. However, this may create maintainability issues and increase storage costs across the multiple VMs.

SWF can scale both horizontally and vertically. Vertical scaling is straightforward in SWF by choosing larger VM configurations. Horizontal scaling requires launching a new VM and registering it with SWF. SWF then acts as load balancer, distributing requests to multiple VMs.

9.5.4 Optimization and Throughput

Our approach to creating a cost model uses benchmark data collected from specific versions of our business process execution framework. Optimizations that reduce the (gas) cost for process execution on blockchain are available. In particular, these optimizations can minimize the storage space required to capture process execution state and to reduce the number of write operations. These improvements can reduce execution cost by around 25%. However, the overall structure of the cost models and cost modelling methodology outlined in this chapter still applies. The only changes are in the values for some of the variables in the blockchain cost model, which can be established by rerunning the cost benchmark tests.

Our cost model calculates the amount of gas that is required to send a particular transaction. The reduction in cost for this transaction leads to an increase in the maximum throughput for transactions of this type. This is because maximum throughput can be obtained by dividing the block gas limit with the gas required for this transaction type. Furthermore, the cost model can inform an analysis of whether a public blockchain can cope with the demand of your business process or application, by calculating the required gas based on the transactions involved.

9.6 Summary

Cost is a critical concern in the design of a software system. The cost of basic compute and storage on public blockchains has a different cost structure than conventional cloud infrastructure but can overall be orders of magnitude more expensive.

To illustrate that, we compared the cost of executing business processes on blockchain with the cost on cloud services, using a large-scale process dataset from industry and an example process from the literature. We demonstrated how to construct and benchmark cost models for both kinds of infrastructure and described experiments that show the cost models performed consistently. The experiments also showed that the cost for business process execution on Ethereum blockchain can be three orders of magnitude higher than on Amazon SWF: for the processes, the average cost per process instance was US$ 1.22 vs. US$ 0.0013 (invoicing), respectively, US$ 1.34 vs. $ 0.0010 (incident management). Given the high volatility of the exchange rate, a cost estimation model that incorporates exchange rate is more important than ever. Our cost model allows to calculate the gas cost per transaction for a given application. On this basis, we have discussed how our approach can be used to build an understanding of the throughput scalability limits for blockchain-based applications. Furthermore, we analysed the impact of different operational workload assumptions on the cost.

If blockchain costs so much more than cloud infrastructure, then why should we use it? The major reason is to benefit from blockchain's trust assumptions.

Blockchains, especially public blockchains, are not controlled by a single party and provide a neutral ground for people and organizations. This can be ideal for multiparty business process execution. The increased cost can be thought of as offsetting the 'cost of distrust' in cloud services across an ecosystem. Another reason is that some blockchain services are much less expensive than conventional services. For example, transaction fees and escrow fees for conventional services can be much higher than on blockchain, and directly incorporating these services into process execution can lead to blockchain being cheaper overall, despite its higher cost for storage and computation.

Finally, we note that cost is often in trade-off with other non-functional properties. We discussed some blockchain-specific factors leading to cost trade-offs for maintainability and scalability.

9.7 Further Reading

This chapter is partly based on our earlier works (Rimba et al. 2017, 2018). More details, especially on the experiments, can be found there.

The Ethereum yellow paper (Wood 2015–2018) defines the specification and the costs of different operations in Ethereum; the experiments reported here are based on the Homestead Draft version.

The incident management case study workflow is adapted from the literature (Object Management Group 2010, p. 18), and we use the process execution approach described in Weber et al. (2016) and the previous chapter. The mentioned optimizations are discussed in (García-Bañuelos et al. 2017).

The performance variation on AWS EC2 has been investigated in the research literature (Iosup et al. 2011; Schad et al. 2010).

The book by Bass et al. (2012) lists many non-functional properties of a software architecture, with cost being one of them.

An online tool to estimate the amount of gas required in Ethereum is available at http://remix.ethereum.org. Live statistics about gas prices on the public Ethereum chain are available on https://ethgasstation.info/.

Chapter 10
Performance

with Rajitha Yasaweerasinghelage

The performance characteristics of blockchains (especially public blockchains) are drastically different from conventional systems. In this chapter we look at how architects can predict the performance of blockchain-based systems, where a blockchain is just one component. We start with a general discussion blockchain performance characteristics, and then describe how to do performance modelling and simulation for specific blockchain-based system architectures.

10.1 Performance Characteristics of Blockchain

Time-related performance requirements are often critical for software systems. There are a number of different kinds of performance requirements, and the two most common are *latency*, which is about how quickly the system responds to a request, and *throughput*, which is about how many requests can be processed within a time period.

Public blockchain platforms have well-known performance limitations for both latency and throughput. The maximum throughput for Bitcoin is estimated to be less than 7 transactions per second and for Ethereum less than 20 transactions per second. This performance is much less than can be achieved with conventional technology using administratively centralized distributed systems. Consider also latency. In a blockchain using Nakamoto consensus (longest chain wins), to confirm a transaction, it needs to be included in a block, which should be endorsed by dependent blocks, known as *confirmation blocks*. Transaction inclusion is probabilistic, so the number of confirmation blocks that one should wait for is a risk-dependent decision. However, on Bitcoin, the average inter-block time is about 10 min, and 6-block confirmation is often used. On the public Ethereum blockchain,

X. Xu et al., *Architecture for Blockchain Applications*,
https://doi.org/10.1007/978-3-030-03035-3_10

the average block generation time is between 14 and 15 s (in July 2018), and 12 confirmation blocks are typically recommended.[1] Individual inter-block times can deviate substantially from those averages.

Clearly, the latency for initial inclusion of a transaction is already higher than for traditional systems, and a large number of confirmation blocks will multiply this delay. Transaction delays can also arise from network delays, the transaction fee offered, the number of transactions being processed, and the strategic decisions made by miners. So, transaction inclusion and commit times can vary widely. Some of these issues are discussed in depth in Section 11.6.

Private blockchain platforms can rely on stronger trust assumptions for nodes than public blockchains, and use consensus algorithms that result in much better latency and throughput. In particular, private blockchains tend not to use Nakamoto consensus, and instead use consensus mechanisms with more conventional commit semantics; when a transaction is included, it is always included, so confirmation blocks are unnecessary. However, even private blockchains tend to have worse latency and throughput than conventional distributed systems.

Regardless, although blockchains often have worse performance than conventional systems, that performance may be perfectly acceptable for some use cases. Once some basic performance threshold is met, often there are diminishing returns for improved performance. The question for architects is: *will basic performance requirements be met by the design of a blockchain-based system?* This concerns all the functions of a blockchain which an architecture uses, be it for storage, computation, communication, or asset management and control as per Chapter 5. For all these functions, read/receive operations can be very fast and with unbounded throughput, but write/send operations are subject to limited transaction inclusion and commit times.

An inability to predict overall performance may itself be a barrier to the adoption of blockchain technology. It is important for designers to be able to accurately predict system-level performance, during the design phase. This allows designers to assess the impact of platform performance limitations on system requirements and to make choices from among the varieties of blockchains discussed in Chapter 3. This includes choices between public and private blockchain, the number of confirmation blocks, and the appropriate integration with off-chain communications and enterprise systems.

Some high-level performance characteristics are known for blockchain platforms, as discussed above. However it is not as widely known how to predict the performance of blockchain-based systems. In Section 9.5.4, we discussed how to obtain throughput estimates from gas price modeling. This chapter shows how we can use architectural performance modelling and simulation tools to predict the latency of blockchain-based systems. We use established tools and techniques but show how blockchain-specific issues can be treated. This includes configuration

[1] http://ethereum.stackexchange.com/questions/183/how-should-i-handle-blockchain-forks-in-my-dapp/203#203.

options such as the number of confirmation blocks and choice of inter-block time. We illustrate the approach using a lab-based experimental study of the incident management system previously introduced in Chapter 9. The method made predictions of median system-level response time, with a relative error of mostly under 10%, making the approach precise enough for capacity planning. We discuss how the approach can be used to support architectural decision-making during the design of blockchain-based systems.

10.2 Architectural Performance Modelling

Architectural models can be used to predict the non-functional properties including latency, throughput, resource usage, and cost. These models can be used by analytical solvers or simulation engines to predict non-functional performance of a system at various stages of the development life cycle.

There are two types of performance models:

- *Analytical performance models* capture performance aspects of the system and serve as input for the analytical solvers. Petri nets (PN), queueing networks (QN), and layered queueing networks (LQN) are examples for common analytical models.
- *Architecture-level performance models* capture key factors influencing the performance of a system. Examples are the Palladio Component Model (PCM); UML profile for Schedulability, Performance, and Time; and Descartes Modelling Language. Architectural models can be either simulated or automatically converted to analytical models. Generally, simulations take a longer time than the solvers to execute but may be more flexible.

In this chapter, we have used the Palladio workbench[2] for architecture modelling of the latency of blockchain-based systems. Palladio is freely available, supports the simulation of architecture models, has a 'UML-like' interface for model construction, and has proven flexibility for extensions such as architectural optimization and new qualities.

10.3 Predicting Latency for Blockchain-Based Systems

To enable latency prediction on the architecture level, we first need to measure latency of individual components. This section first describes our approach to benchmarking transaction inclusion and commit times for blockchain. We then describe our approach for system-level performance modelling, using the method

[2] https://www.palladio-simulator.com.

for business process execution on the blockchain described in Section 8.2. These performance models are configured using the benchmarking results.

10.3.1 Benchmarking Transaction Inclusion and Commit Times

A key parameter for our performance model is the *transaction commit time*: the time taken from submitting a transaction until we have sufficient confidence that the transaction has been successfully included in the blockchain. We need to benchmark this in a representative deployment of the blockchain to be used by the client application. The idea is simple: start the clock when we submit a transaction, and stop the clock when the transaction is 'committed'. In the Ethereum blockchain we use in this chapter, a transaction is committed when the broadcasting node receives a sufficient number of confirmation blocks after receiving a block which includes the transaction, as mentioned earlier. If one block is enough as confirmation, we call this *transaction inclusion time* instead. The total time will depend on the transaction propagation time, inter-block time, transaction inclusion probability, block propagation time, and the number of confirmation blocks. Our benchmark measurement abstracts from these details to create a transaction inclusion time distribution that we use in our performance model. Our benchmark measurements also include latency overhead for our trigger code and the communication between the trigger and the Ethereum node. However, this overhead is in milliseconds range, compared to the seconds for inter-block times, so it is not significant; and in any event, client applications using the blockchain encounter similar delays.

As discussed previously, the number of confirmation blocks is a design choice for client applications using a blockchain. Although 12 confirmation blocks are often recommended for the public Ethereum blockchain, the 'right' number depends on the business risk involved in the transaction and on other trade-offs with latency.

To demonstrate the approach, we ran benchmarks on a private instance of the Ethereum blockchain. We used a private deployment to prevent flooding the public Ethereum blockchain, to reduce our cost, and to be able to vary inter-block time. We used one virtual machine to deploy the trigger and a go-Ethereum (geth) full node with mining disabled. The mining node was deployed on a different virtual machine. This situation would mimic practical deployment to some degree: each organization would deploy their own full node and trigger in a virtual machine controlled by them, whereas miner nodes are operated on separate machines. Both virtual machines run on one Intel(R) Xeon(R) CPU E5-2697 v3 at 2.60 GHz core each. The virtual machines were located in the same data centre and had a LAN connection. The trigger was implemented in Node.js version 4.2.6 using Ethereum JavaScript library (web3) version 0.15.3. Geth version 1.5.4-stable was used, and

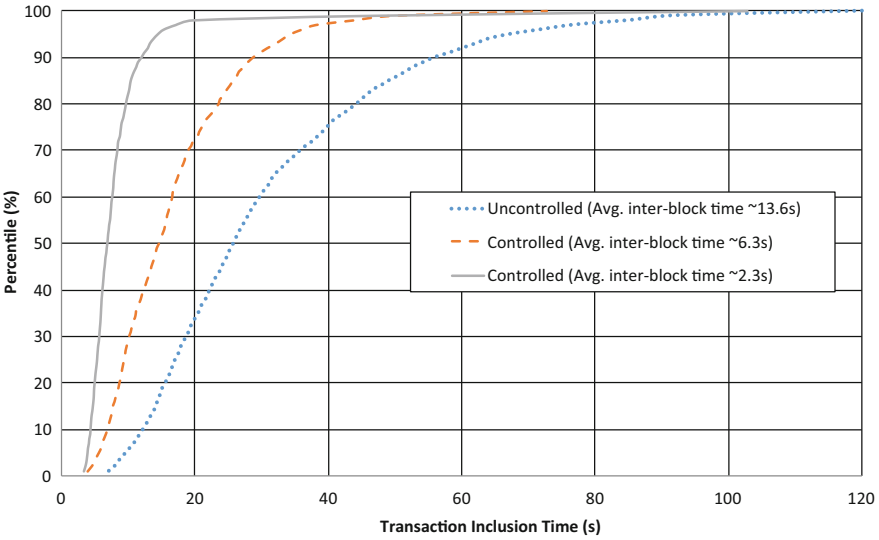


Fig. 10.1 Transaction inclusion time measured on Ethereum (cumulative). © 2017 by the Commonwealth Scientific and Industrial Research Organisation, reprinted with permission

the trigger was configured to use remote procedure call (RPC) communication[3] to interact with the geth node.

For benchmarking latency, we submitted many transactions as follows. A script invoked the trigger API, which submitted the transaction. The trigger then listened to the blockchain for the announcement of a sufficient number of confirmation blocks after observing a block including the transaction and forwarded the result of successful inclusion/commit back to the script. The script initiated the next transaction directly afterwards.

As a baseline, we report here the observations of transaction inclusion time (i.e. where sufficient confidence of inclusion is judged to have occurred on seeing the transaction in a block, as defined above). We ran the experiment on a private blockchain, where we varied inter-block time, by either controlling the complexity mechanism or using the default implementation (uncontrolled). The mean *inter-block time* of the uncontrolled blockchain was 13.6 s. In two settings of controlled private blockchain settings, we measured mean inter-block times of 2.3 and 6.3 s.

For each of the three settings, we measured *transaction inclusion time* across 1000 transactions. The results are shown as cumulative distributions in Fig. 10.1. While median transaction inclusion time was 25.8 s on an uncontrolled private

[3]Note that the communication latency between geth and other components can be reduced by using inter-process communication, as discussed by García-Bañuelos et al. (2017). Note also that IPC requires fully asynchronous communication between these components, and the decision should be made before major refactoring of code becomes necessary to implement that.

blockchain, it was 6.91 and 14.65 s, respectively, for the two controlled private blockchains with 2.3 and 6.3 s mean inter-block times. To understand these times, note that miners roughly operate as follows: when they receive or mine a block, they gather a set of transactions from the pool for inclusion in the next block and then try to solve the proof-of-work puzzle. For a transaction to be included, it therefore needs to be in the pool already when work on the next block begins. Therefore transaction inclusion time is always higher than inter-block time. It should also be noted that median transaction inclusion time would be higher on public blockchains, because of additional network delays and strategic transaction inclusion by miners. For inclusion times on the public Ethereum blockchain, refer to the experiments in Section 11.7.

10.3.2 Blockchain-Based System Performance Modelling

The benchmark tests above tell us the latency for an individual transaction. We can use those performance benchmarks to build higher-level predictive models of latency for an entire blockchain-based system. This section describes how, illustrated using a blockchain-based system generated using the model-driven development method described in Section 8.2 and based on the incident management process model shown in Fig. 9.2. The performance models refer to specific aspects of the incident management process; therefore we briefly explain it first. There are four issue resolution stages: account manager, first-level support, second-level support, and developer support. When a customer reports an issue first, the account manager requests a problem description and attempts to solve the issue. If the issue is solved directly, the account manager provides the solution to the customer. Otherwise, the account manager asks first-level support, and if first-level support cannot solve the issue, they ask second-level support and so on. At each stage, if someone finds the solution, they give feedback to the upper level, and finally the account manager explains the solution to the customer.

For performance prediction, we model the blockchain as a single component, from the perspective of the client application. So, we do not model the details of the blockchain mining network, node intercommunication, or consensus algorithm. All of these factors are aggregated in our abstract model and measurements. The client application interacts with the blockchain through a local blockchain node, and we model the resource and performance characteristics of this local blockchain node running as a component. In the architecture of a scalable client application, one may need to operate multiple blockchain nodes, each independently participating in the blockchain system; in such cases, we would model those as multiple deployed instances of the blockchain client. Note that these blockchain clients do not need to be resource-intensive mining nodes attempting to create new blocks on the blockchain. Instead, it is enough for these nodes to be full nodes, submitting and observing transactions and blocks on the blockchain network.

Component Repository Model

In our model-driven development method, off-chain business process systems interact with the blockchain through trigger components. Figure 10.2 shows an example Palladio Component Model (PCM) of a trigger component connected to an Ethereum blockchain client node (using the Ethereum geth client). They are modelled as two components, each exposing a relevant interface that defines various operations. These operations correspond to actions in the example incident management business process. The trigger interface also provides a *createInstance* operation, which creates an instance of a process monitor by invoking the factory smart contract for the business process, pre-configured on the blockchain. The trigger translates API calls into corresponding blockchain transactions and submits them for execution on the blockchain through the locally deployed Ethereum client.

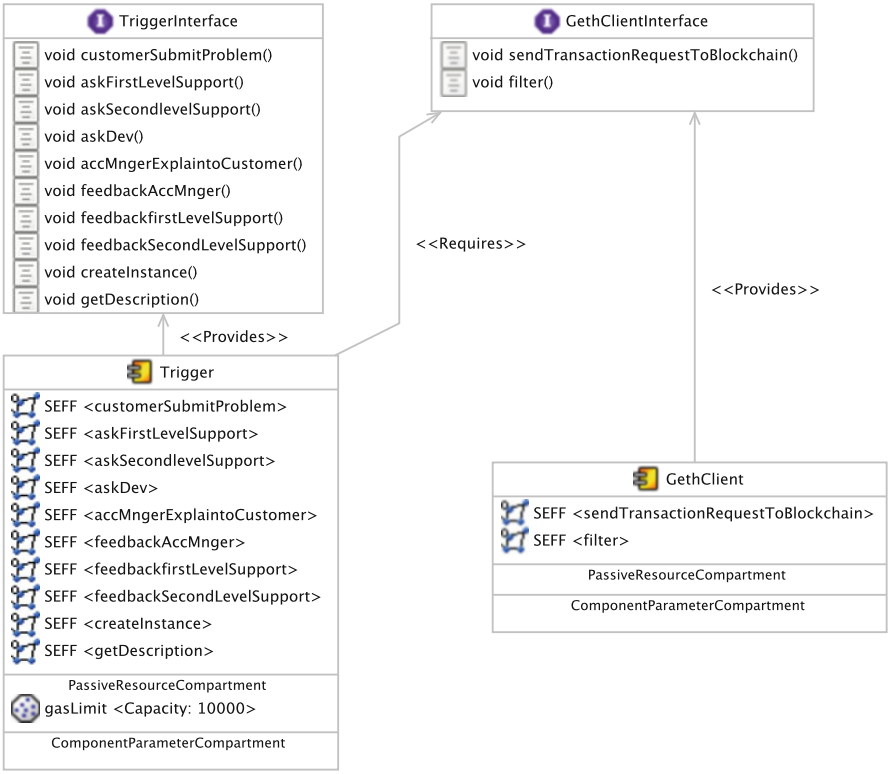


Fig. 10.2 PCM repository diagram showing connected components and the operations provided by their interfaces. © 2017 by the Commonwealth Scientific and Industrial Research Organisation, reprinted with permission

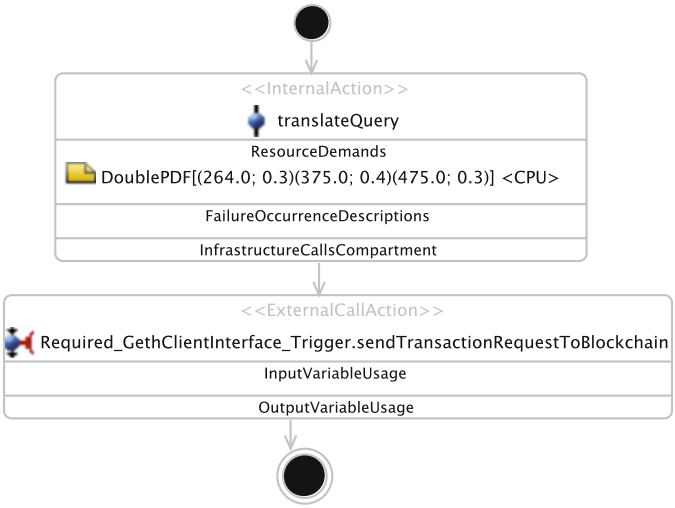


Fig. 10.3 RDSEFF diagram of operation transaction. © 2017 by the Commonwealth Scientific and Industrial Research Organisation, reprinted with permission

Resource-Demanding Service Effect Specifications (RDSEFF)

The PCM repository diagram is a model of the components, interfaces, and their relationships. We also need to model the non-functional behaviour of component operations. In PCM, each component operation's behaviour is specified in a resource-demanding service effect specification (RDSEFF). The model describes how each operation translates an API call to a blockchain transaction and uses an external action to forward the transaction to the blockchain node, as illustrated in Fig. 10.3. The resource utilization of each component is configured as a probability distribution function (PDF) constructed using benchmarks as described in Section 10.3.1. Each operation is benchmarked and modelled separately to account for variation in the operations and to demonstrate the capability of modelling their different behaviours. The manual steps in the process (such as operator resolution time) must be separately benchmarked for inclusion into the model, but this is not dealt with in this chapter.

Usage Model

To simulate the execution of the system, we specify a usage model that captures representative use of the system. Our example usage model in Fig. 10.4 reflects process flow in our example business process for incident management. The points of variation are for the optional branches of the process. For the purpose of our laboratory experiments, we assumed that at each stage of incident response (except

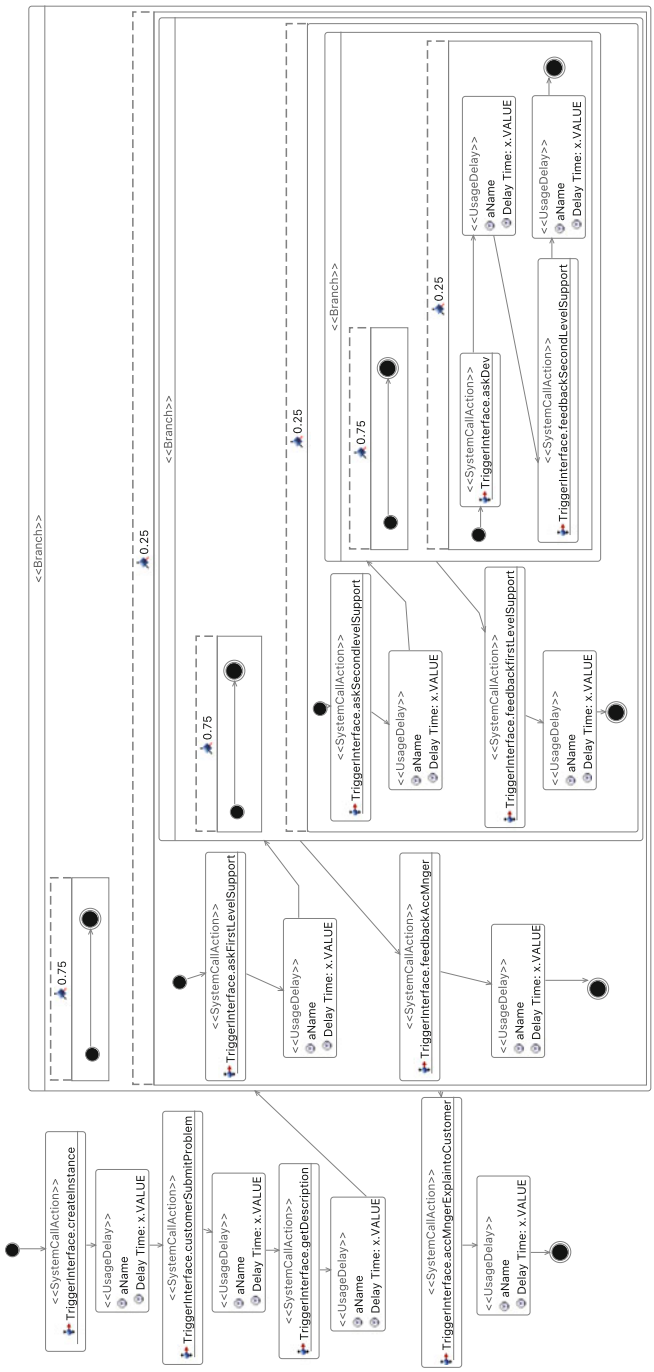


Fig. 10.4 PCM usage model diagram of incident management business process. © 2017 by the Commonwealth Scientific and Industrial Research Organisation, reprinted with permission

for the final developer stage), 75% of issues received were resolved in that stage. The final developer support stage resolves every request. Figure 10.4 shows the branching probabilities used to represent this behaviour.

Here we show a usage model as a single scenario with branching probabilities. Variation in the possible resolution times, e.g. due to randomness in the path taken, is explored through multiple simulation runs. However, it would also be possible to examine multiple usage scenarios separately, each using different probabilities or execution/resolution times. This could be done to drill down onto specific issues or opportunities regarding the design of the business process.

10.3.3 Using Simulation for System-Level Latency Predictions

Now we show the simulation results from our performance model, and show how accurate the predictions (based on micro-benchmarks and architecture models) are by comparing them to macro-level measurements from an implementation. In our approach, our system has to use the same or similar underlying blockchain platform for which we have collected transaction performance benchmarks. So our implementation here uses the same private Ethereum environment as in Section 10.3.1. Here we only measure the business process's end-to-end latency, for example, incident management process described in Chapter 9. Further details are discussed below.

A synthetic workload was generated which follows the same 75% resolution rate at each stage as in Section 10.3.1. Trigger operations were invoked by HTTP requests using an external python script, which was deployed separately, and we measured the time from initializing a process instance until observing its completion. Between the completion of one instance and the start of the next instance, the script waited for 1 s. The experiment was run for 1000 times (creating 1000 process monitor instances), which took approximately 20 h.

For simulation, the SimuCom simulation engine was used for executing the PCM model and ran the same number of scenario executions. SimuCom is the standard simulation engine for PCM model simulation.

The measured and predicted latency results are shown in a boxplot diagram in Fig. 10.5, where the box indicates median values in red and the first and third quartiles as upper and lower bound of the box. The predictions appear largely accurate when compared visually, and statistically the simulation predicted the mean latency of the process scenario with a relative error of 1.6%. The measured mean latency of the process was 136.29 s, and the simulation predicts the mean latency as 134.08 s where the standard error of mean (SEM) is 1.27 and 1.07, respectively. For many applications, 95th and 99th percentiles are significant measures when considering the latency and the skewness of the distribution. The PCM model predicted the 95th and 99th percentiles with a relative error of 9.4% and 11.5% accuracy. Errors in predicted maximum and minimum are, respectively, 7.62% and 16.89%.

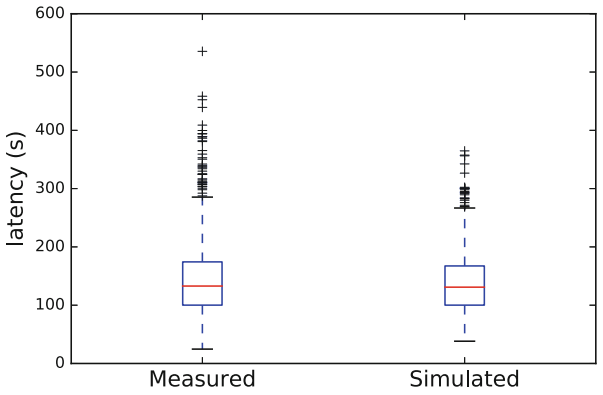


Fig. 10.5 Boxplot diagrams of measured and simulated scenario latency—measured median is 132.83 s, simulated median is 130.93 s, relative error of median is 1.42%, and relative error of 95th percentile is 14.6%. © 2017 IEEE. Reprinted, with permission, from Yasaweerasinghelage et al. (2017a)

Applying Simulation to Other Systems

The architectural performance and simulation approach we have used is largely consistent with the previous body of work in this field which has been applied to a variety of application systems. Our approach should similarly apply to other systems. However, when doing so, a few aspects deserve attention:

- The transaction inclusion time benchmarks we showed in Section 10.3.1 are specific to our customized version of the Ethereum client. In particular, our experiment setting had no significant network delays for transaction or block propagation among peers, and there is no occurrence of *uncles* (short-lived alternate competing recent histories). These factors are likely to affect transaction inclusion and commit times. We recommend benchmarking end-to-end latency of transaction commit time in the target blockchain platform in order to account for all sources of delay and variation in transaction inclusion.
- Similarly, our experiments on Ethereum use Nakamoto consensus and proof-of-work. We expect our modelling approach would be usable for other consensus mechanisms, after benchmarking transaction inclusion and commit times in those systems. Our general approach would be applicable in blockchains using classical distributed consensus algorithms, but the stronger transaction commit semantics supported by those algorithms means that confirmation blocks would not be required.
- Our focus in this chapter is on latency, not throughput or scalability. We have therefore benchmarked latency and evaluated predictions under low demand. In our experiments, we observed low CPU load, so assume that CPU utilization did not impact latency. In real-world situations, latency is affected by high demand, resource bottlenecks, and architectural mechanisms (e.g. load balancing) used for scalability. We expect that for a particular use case, if a representative load can be

used on a representative deployment of a blockchain, then latency benchmarking could be performed as we have described in this chapter.
- In a blockchain-based system, in addition to CPU, network, and disk, there are other resources. We have not modelled smart contract gas consumption, gas limits, and public blockchain transaction fees, although these may able to be modelled as passive resources in Palladio.

10.4 Architectural Decision-Making

Design alternatives can be evaluated by predicting latency in example scenarios. This lets us explore *what-if* questions in architectural decision-making. Here we focus on latency and how it is impacted by architectural changes.

10.4.1 Choice of Inter-Block Time

In a public blockchain, the target inter-block time is fixed. However, in private blockchains, it can be varied as a design choice. This reduces transaction inclusion and commit times, which can reduce system-level latency. When evaluating inter-block time alternatives, we use the same system-level models and only change the transaction inclusion time parameters.

We conducted an experimental evaluation of the accuracy of our simulation for various transaction inclusion times, on a private blockchain. To illustrate this, the results for transaction inclusion time for a 2.3-s inter-block time is shown as a boxplot in Fig. 10.6. The relative errors are still good: the relative error of median was 9.4%, and the relative error of 95th percentiles was 8.5%.

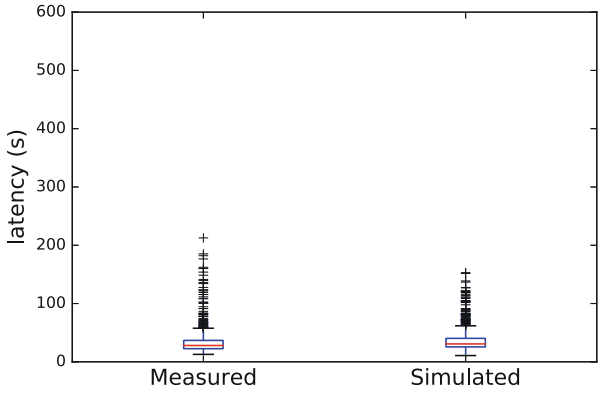


Fig. 10.6 For 2.3-s average inter-block time: median time (measured 28.1 s, simulated 30.7 s), relative error (median 9.4%, 95th percentile 8.5%). © 2017 IEEE. Reprinted, with permission, from Yasaweerasinghelage et al. (2017a)

The median latency dropped from about 130 s to about 29 s. Whether this is acceptable for the system design depends on the system requirements, but in any event the simulation results would provide a reasonable basis for making this decision before the system is implemented.

10.4.2 Choice of Number of Confirmation Blocks

Blockchain-based systems using Nakamoto consensus can be vulnerable to double-spending attacks. This vulnerability can be reduced by increasing the number of confirmation blocks. However, this introduces additional latency to the system.

To illustrate this, we show results from an experiment using 12 confirmation blocks in Fig. 10.7, using a controlled blockchain with a mean inter-block time of about 2.3 s. The measured median process latency was 152 s vs. the simulation prediction of 164 s. The relative error of the median prediction was 7.9%, and the relative error of 95th percentile was 12.3%.

The latency for 12 confirmation blocks of around 160 s is much higher here than the latency for 1 confirmation block of around 29 s. Whether this additional delay matters or not is a question for the system requirements. The key point is that the simulation model provides a reasonable basis for exploring the performance consequences of these design options before they are implemented.

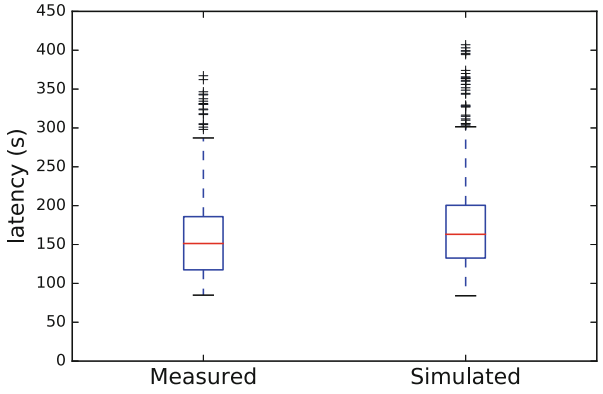


Fig. 10.7 12 confirmation blocks: median time (measured 152 s, simulated 164 s), relative error (median 7.9%, 95th percentile 12.3%). © 2017 IEEE. Reprinted, with permission, from Yasaweerasinghelage et al. (2017a)

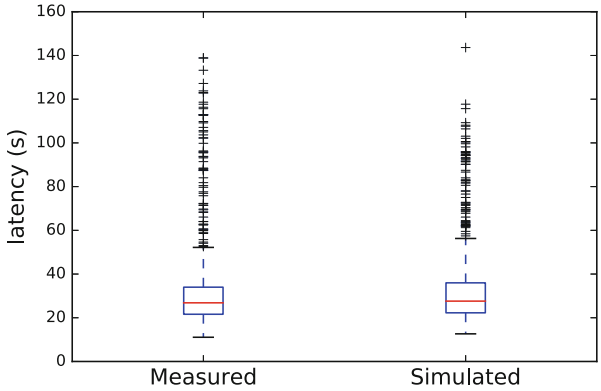


Fig. 10.8 Measured and simulated latency of modified business process. Median time (measured 26.8 s, simulated 27.6 s), relative error (median 2.9%, 95th percentile 0.3%). © 2017 IEEE. Reprinted, with permission, from Yasaweerasinghelage et al. (2017a)

10.4.3 Process-Level Changes

The previous two simulation scenarios have involved changes to the configuration of the blockchain platform. However, we can also use simulation to predict the performance impacts of process-level changes. Instead of changing the performance benchmark parameters, we instead change the Palladio usage model. There are many kinds of possible process redesign such as task elimination, process integration, or task composition. All of these can be modelled by changing the workflow. Most business process control flow patterns can be directly translated to Palladio Component Model patterns.

To explore this, we experimented with a changed process model, where the account manager assigns issues directly to second-level support (skipping the first level) in 5% of cases. We used a private Ethereum blockchain with a mean inter-block time of 2.3 s. The results are shown in Fig. 10.8. The median process latency was measured as 26.8 s vs. 27.6 s from simulation. The relative error of median was 2.9%, and the relative error of 95th percentile was 0.3%. So there are some small performance improvements. However, a full cost-benefit analysis would depend not just on latency considerations but also on the relative cost and resource utilization of various support tiers.

10.5 Summary

This chapter started with a broad discussion of performance and blockchain's impact on application performance. We have then shown how to predict the latency of blockchain-based systems using architectural performance modelling

and simulation. In the approach, we benchmark transaction inclusion time for the blockchain platform being used, then include those performance benchmarks in a system model. The blockchain is treated as a black-box component, and blockchain transactions are connected to operations at the application level. In experiments, the system-level latency predictions have a relative error of mostly under 10%, which means that the approach is precise enough to use to evaluate designs before they are implemented. The importance of this for architects is that a wide range of architectural alternatives can be analysed. Some of these decisions are about blockchain-specific issues, such as inter-block time or the number of confirmation blocks. Other design decisions, such as possible business process changes, are system-level design options but are impacted by latency arising from the blockchain-related factors.

We have focussed on latency in this chapter, because it is mostly under the control of designers of blockchain-based systems. The throughput of a blockchain is mostly governed by the initial choice of blockchain platform and its block configuration, and this might not be easily changed. However, latency and performance more generally are not the only non-functional properties that are important in the design of systems. Trade-off decisions need to balance predicted latency impacts with the predicted impacts to other qualities.

10.6 Further Reading

This chapter is partly based on our earlier works (Yasaweerasinghelage et al. 2017a,b). Additional details on experiments can be found there.

Architectural models can be used by analytical solvers or simulation engines to predict non-functional performance of a system at various stages of the development life cycle (Brunnert et al. 2015). *Analytical performance models* capture performance aspects of the system and serve as input for the analytical solvers. Petri nets (PN) (Molloy 1982), queueing networks (QN) (Bolch et al. 2006), and layered queueing networks (LQN) (Franks et al. 2009) are examples for common analytical models. *Architecture-level performance models* capture key factors influencing the performance of a system. Examples are the Palladio Component Model (PCM) (Becker et al. 2009); UML profile for Schedulability, Performance, and Time (Xu et al. 2003); and Descartes Modelling Language (Kounev et al. 2014).

In this chapter, we have used the Palladio workbench (Becker et al. 2009) for architecture modelling of the latency of blockchain-based systems. Palladio's extensions allow for architectural optimization (Koziolek et al. 2011; De Gooijer et al. 2012) and can be enhanced with new qualities (Willnecker et al. 2014).

For live statistics about the public Ethereum chain, including inter-block times and the influence of the gas price on transaction inclusion times, see https://ethgasstation.info/ and https://ethstats.net/.

The blockchain-based system we used in our experimental evaluation is a business process system using the approach from Weber et al. (2016), which also measured transaction inclusion time and utilized the incident management exemplar use case we use here. The same system was also discussed in the previous two chapters, albeit not the experiments mentioned above. García-Bañuelos et al. (2017) discuss gas cost optimization for such a system and include a large-scale *throughput* experiment.

Chapter 11
Dependability and Security

with Ralph Holz, Vincent Gramoli, and Alex Ponomarev

In this chapter, we discuss dependability and security aspects of blockchain-based applications and analyse how the different properties of dependability and security relate to these applications. As is the case throughout the book, our viewpoint for the discussion in this chapter is that of a system architect or developer using blockchain as a component. We thus analyse how blockchains impact the dependability and security of systems built upon them, in part with studies using observations from the mainstream proof-of-work blockchains Ethereum and Bitcoin. As such, we are not going into the details of cryptography and security infrastructure of blockchain platforms.

Dependability and security are tightly interlinked. According to the widely accepted taxonomy of Avizienis et al. (2014), dependability and security are comprised of six attributes as shown in Fig. 11.1. The first five sections of this chapter give an overview of the influence on dependability and security attributes of blockchain as a component within a multiparty system.

We then focus on the availability of functions that such systems need, in particular transaction inclusion, and how they may be adversely impacted by a number of factors. When viewing blockchain as a component for data storage, communication, or code execution, whether a transaction is included or not can largely be equated to write/send availability.

Finally, in Section 11.7, we discuss issues around aborting and retrying transactions—a functionality that is not provided by blockchain client software today.

X. Xu et al., *Architecture for Blockchain Applications*,
https://doi.org/10.1007/978-3-030-03035-3_11

Fig. 11.1 Attributes of security and dependability. © 2004 IEEE. Reprinted, with permission, from Avizienis et al. (2014)

11.1 Confidentiality

Confidentiality means that unauthorized disclosure of information does not take place. This is usually harder to establish in blockchain-based systems, because the default is that information is visible for everyone in the network. Information can be encrypted: asymmetrically with a particular party's public key, so that only this party can decrypt it, or symmetrically with a shared secret key, so that the group of parties with access to the secret key can decrypt it. The latter case requires a secure means of exchanging the secret key, typically off-chain. However, once information needs to be processed by smart contract methods, this information needs to be decrypted. This is because smart contract code runs on all nodes of the network, and thus any of them needs to be able to process the input data. The ability for anyone to execute smart contracts is required to achieve consensus on the outcomes of smart contract execution. Embedding keys within a smart contract would reveal the key to all participants.

As discussed in the supply chain use case in Section 4.1, commercially sensitive data can be at risk if it is shared on a blockchain, even if pseudonyms are used and even if encryption is used. Private and permissioned blockchains can provide read access controls, but this will not provide commercial confidentiality between competitors using the same blockchain.

There are interesting technologies on the horizon, which could alleviate some of these pain points. For instance, zero-knowledge proof methods like *zk-SNARKs* can be used to hide the contents of a transaction, while still allowing independent validation of the integrity of that transaction. Current implementations are limited to hiding simple transfers of cryptocurrency, but in the future the same could be achieved for more sophisticated transactions. As for computation on encrypted data, that is the goal of techniques like *homomorphic encryption* and *confidential computing*. However, such approaches have not been utilized for smart contracts as yet, in part due to their significantly increased computational requirements over regular computation. Alternatively, authorized 'witnesses' could have special access to the data. These witnesses could be certifying agencies or consumer group advocates. The data would be encrypted using the witness' public key, so that only the witness can decrypt it. The witness can then pass on the provenance information to interested parties, but not share information that is commercial in confidence. How the data is to be encrypted and stored would be part of the smart contracts created for various supply chain events and, as such, can be customized for different scenarios and supply chains.

11.2 Integrity

Integrity is the absence of improper (invalid or unauthorized) system alterations and is a key attribute for blockchains. Once a transaction is included in a blockchain and committed with sufficiently many confirmation blocks, it becomes part of the effectively immutable ledger and cannot be altered. This also applies to smart contracts: their bytecode is deployed in a transaction and thus is subject to the same integrity guarantees; and invocation of smart contracts happens through transactions as well. The key integrity property of Bitcoin is that addresses cannot spend money they do not have. Ethereum's integrity property is more complicated, because it requires the correct operation of a Turing complete smart contract programming language. However, for client applications, Ethereum provides significant power by allowing user-defined integrity conditions to be implemented as checked preconditions and defined behaviours in smart contracts.

Blockchain emerged to support a cryptocurrency, and so it is unsurprising that integrity is a key dependability attribute, because integrity is the key dependability attribute for commercial computer security. The seminal work on this topic is the Clark–Wilson security policy model, and blockchains are broadly consistent with its requirements. Smart contracts can implement Clark–Wilson's transformation procedures to generate and update internal data or other smart contracts that realize Clark–Wilson's constrained data items. Blockchains natively create the log required by Clark–Wilson for reconstructing operations. Finally, blockchains use a kind of separation of duty through the replicated validation performed by all mining nodes.

Ethereum smart contracts are written in a Turing-complete programming language. This makes it more difficult to verify that the smart contracts correctly implement required integrity properties. Formal verification techniques can be used, but these can be costly and time-consuming in practice. A lighter-weight approach is to use a smart contract language with strong typing mechanisms, which can help programmers support integrity. The Pact language on the Kadena blockchain[1] is an example of that approach. Some blockchains, such as Kadena and Corda,[2] avoid the use of Turing-complete smart contract languages for this reason, and instead use less-expressive domain-specific languages that can be automatically checked.

High integrity and non-repudiation are not always ideal. For example, sometimes historical data must be deleted or changed. If a vexatious or improper registry entry has been created, a court may order the registrar to change the registry to remove that entry, 'as if it had never been created'. This is not technically possible on many blockchain platforms. Similarly, this may create problems for blockchains that have been 'poisoned' by illegal content. Some blockchains have been proposed to deal with this challenge, but there is not yet widespread acceptance and adoption of good solutions.

[1] http://kadena.io/.

[2] https://www.corda.net/.

11.3 Safety

As defined by Avizienis et al. (2014), safety means that using a system does not lead to catastrophic consequences on the users and the environment. The use of blockchain technology does not directly pose specific risks in this regard, when compared to other components in distributed systems. There may be economic and environmental risks from investments in ineffective blockchain mining strategies. By using non-mining nodes, private or permissioned chains, and/or alternative consensus mechanisms, these risks can be mitigated. If a blockchain is used as a component in a safety-critical application, then failures of integrity, confidentiality, availability, or other dependability attributes may have consequences for safety. However, this is a prevalent issue of safety-critical system engineering. A noteworthy difference exists when the cryptocurrency or token features of a public blockchain are used, in which case an organization or user is exposed to the monetary risk of loss or devaluation of the cryptocurrency and tokens. If this risk can bankrupt the organization or users, it may lead to situations that could be seen as catastrophic. With respect to cryptocurrency, a difference to regular internet-related flow of monetary assets lies in the fact that there is no additional safety net. No banks will stop attacks on your Bitcoin wallet or reimburse your losses. In most cases of theft, lost crypto-coins or tokens remain unrecoverable.

An alternative, informal definition of safety by Lamport (1977) states that some 'bad thing' does not happen during execution. Alpern and Schneider (1985) later formalized this definition with regard to discrete execution states of programs but did not formalize what a 'bad thing' might be due to the inherently informal nature of this concept. Examples of safety properties mentioned in the above sources are mutual exclusion of concurrent processes, deadlock freedom, partial correctness, and first-come-first-served execution.

This perspective is indeed interesting when considering blockchain. When considering a public blockchain network execution itself as the program, discrete states are almost meaningless: the states of different nodes around the world are only very loosely synchronized, and substantial differences between, say, the current transaction pools of set of nodes can be expected. Considering the committed blocks in a blockchain (e.g. the current Ethereum blockchain minus the most recent 11 blocks), discrete execution states become a valid model. Concerning the above-quoted examples, concurrent processes and *mutual exclusion* are a non-issue (since the execution has been sequentialized).

Deadlocks on the application level can exist as in any other program, be it on the smart contract level or off-chain programs. It might be easier to avoid deadlocks in blockchain-based applications, since smart contracts can be used as a neutral mediator, which handles all resources (e.g. cryptocurrency in exchange for tokens), instead of distributed processes responsible for different resource types.

Though not mentioned in the early literature, *livelocks* or infinite loops in a smart contract would, in the case of Ethereum, be resolved by the platform itself: each invocation has a limited amount of gas available. If the execution does not terminate before the gas has been consumed, it is aborted.

Partial correctness on the application level is not impacted by blockchain. However, correctness of the execution when computing a new block is higher: in many public blockchains, all full nodes verify each newly announced block by checking digital signatures and hashes and executing all transactions. If the results—e.g. of a smart contract invocation check—are inconsistent, the new proposed block is discarded.

If *first-come-first-served* is required, typically that requires special consideration on blockchain. In Bitcoin, strict ordering of transactions can be established by consuming an output of one transaction as input of another transaction—the second is only valid once the first has been included, although both transactions can be included in the same block. Similarly, Ethereum transaction nonces can be used to ensure ordering of transactions, but this feature is only available for transactions originating from a single account. A smart contract can ensure ordering to a degree, e.g. if the order can be prescribed. For scenarios where neither of these options suffice, e.g. open bidding processes, off-chain mechanisms might be required to ensure fair processing. Generally, the inclusion of a given transaction is not guaranteed by blockchain protocols, let alone in any particular order. This issue of availability from the viewpoint of an application will be discussed in depth in Section 11.6.

11.4 Maintainability

Maintainability refers to a system's amenability to undergo modifications and repairs. In blockchain-based systems that use smart contracts, this is harder to implement for the smart contracts than in regular distributed systems. This is because smart contracts comprise code that regulates the interactions between mutually untrusting parties; trust is derived from the fact that the code cannot be changed easily. Consider an example where an organization has established trust in the code of a particular contract and verified that it implements the agreed rules for handling cryptocurrency. If others can change the code without that organization's knowledge or consent, any trust in the code would be void. Although the code of an Ethereum smart contract cannot be changed, the current state of variables within that smart contract can be updated by invoking its methods. In particular, these variables may refer to other smart contracts. This mechanism provides a kind of indirection that allows the dynamic updating of smart contract code, through mechanisms like the Contract Registry Pattern (Section 7.4.1). However, support for this kind of updating must be specifically provisioned ahead of time.

Finally, changes may be made to a blockchain-based system not by changing the data stored on a blockchain but instead by changing the interpretation of data on the

blockchain. As an extreme example, a client application might choose to not honour all data previously written to the blockchain under some previously acknowledged addresses. Instead, the client could in principle re-create all required data on the blockchain under some new address. A distinctive benefit of blockchain-based systems is that there is no single party with control of the system. However, this inherently creates challenges for governance: the management of the evolution of blockchain-based systems. Changes may be made to correct defects, add features, or migrate to new IT contexts. However, in a multiparty system with no single owner, managing these changes is more like diplomacy than traditional risk management or conventional product management. Lessons may be drawn from governance in open-source software, which face similar development challenges. However, the governance of a blockchain is not just a software development problem—it is also a deployment and operations problem. For both public and private blockchain systems, key stakeholders include the users of the blockchain, software developers with moral or contractual authority over the code base, miners or processing nodes in the blockchain ecosystem, and government regulators in related industries. There are still lessons being learned about who the key stakeholders for blockchains are. For instance, the 2016 hard fork of Ethereum in response to the DAO controversy made it apparent in hindsight that digital currency exchange markets are a key stakeholder for public blockchain systems. (The market initially provided by the Poloniex exchange for trading the unforked 'Ethereum classic' digital currency has supported the ongoing operation of that blockchain, which might have otherwise failed to continue to be viable.)

It is unknown how to best perform governance for blockchains and blockchain-based systems. How should relevant stakeholders influence and manage changes to the software and the operational infrastructure for blockchains and blockchain-based system when there might be no central owner and where the blockchain platform might be serving many purposes for different stakeholder groups?

11.5 Availability and Reliability

According to Avizienis et al. (2014), availability is the readiness for correct service, whereas reliability is the continuity of correct service. More specifically in our context, availability concerns the users or dependent systems' ability to invoke functions of the system, whereas reliability refers to receiving consistently correct outcomes from those invocations.

For blockchains, there are scenarios in which the distinction between reliability and availability can be blurred as there is no globally specified time by which a transaction should complete. If a blockchain system never includes a transaction (perhaps because other connected nodes ostracize that transaction, address, or interface node), that will be both an availability and reliability failure of the blockchain system from the perspective of a client application. However, if a transaction is initially included in some block, that does not guarantee that block will

be recognized as being part of the blockchain in future. One could take the following view: first an application designer can specify a number of confirmation blocks by which they will regard a transaction to have been committed. If a fork happens that invalidates transactions thought to be committed, the system will have had a reliability failure, because a transaction thought to have been committed will have turned out not to be. Alternately, if a fork affects less than the specified number of confirmation blocks, the system may experience enough delay to have an availability failure.

The operation of public blockchains can involve hundreds or thousands of independent processing nodes. Each node holds a full replicated instance of the blockchain transaction history and can operate for users as a transaction interface to the blockchain network. Because of this massive redundancy, naively we might expect that a blockchain system has extremely high availability. We can assume that local components of a blockchain-based system are connected to a local full node on the blockchain network. Submitting a transaction to a blockchain network is done through the local full node, which broadcasts that transaction to all nodes it is connected to. The availability of a locally reachable full node is thus heavily reliant on the organization operating a blockchain-based system. The more complex question is: how certain can one be that the transaction is included in a block and committed, in a timely manner? We address this question in the next section in detail.

Transactions deploying smart contracts or invoking their methods add a further level of complexity. First, they are subject to more parameters, like current gas limit, that may impact their successful inclusion. Second, they utilize more complex functionality of the network and thus rely on the network sharing the same accepted norms about this functionality with the system. For instance, parts of the network may change to not accept certain commands present in compiled smart contracts. If the blockchain-based system is unaware of the change, it might attempt to use these commands, and its contracts might get rejected, or method calls might be terminated unexpectedly. Again, we discuss these issues in more detail in the next section.

Finally, we note that the well-known *CAP theorem* indicates that there is inevitably some trade-off between consistency, availability, and partition-tolerance for distributed databases. As described above, blockchain platforms sacrifice traditional notions of consistency, but strive for availability and partition-tolerance.

11.6 Variation in Blockchain Transaction Inclusion

Blockchains are distributed systems, and so the states of different parts of the system are inevitably different. Different nodes will hear about new pending transactions and new blocks at different times. There is also variation in how long it takes for the system to commit transactions in the ledger. For public blockchains like Bitcoin and Ethereum that use Nakamoto consensus, there is much greater variation in transaction inclusion time, which is exacerbated by the probabilistic nature of transaction inclusion.

In fact, there can be so much timing variation that it can impact core dependability attributes. If transactions takes *too* long to be included, they will violate *latency* or service availability requirements. Integrity can also be impacted if transaction reordering occurs because of the probabilistic nature of Nakamoto consensus. This section explores these issues in some detail, for both Bitcoin and Ethereum.

11.6.1 Variation in Bitcoin Transaction Commit Time

In this section, we explore the factors that impact Bitcoin commit time and show that reordering of transactions play an active role.

A peculiarity of Bitcoin is the way transactions are linked: they transfer currency from a number of source addresses to a number of destination addresses. Recall from Section 2.1 that transaction outputs become the inputs of new transactions. If the sum of the outputs is less than the sum of the inputs, this is interpreted as an additional output that pays a fee to the miner who mines the block containing this transaction. This acts as an incentive for miners. As a result, miners tend to optimize block creation by preferring transactions with higher fees. The transaction fee is often the only variable that client software asks Bitcoin users to choose consciously when creating a new transaction.

However, transactions can also experience delay due to other factors. An important one is that transactions must arrive (roughly) in order, for a node (and the network) to be able to process them fast. Incoming transactions are handled by the so-called mempool. If the referenced input transactions (the 'parents') are yet unknown, a miner will delay the inclusion of the new transaction—it is then a so-called 'orphan'. Miners may choose to keep orphans in the mempool while waiting for the parent transactions to arrive, but they may also expunge orphans after a timeout they choose. A second factor that could come into play, albeit one that only experienced users will set, is so-called locktimes: a transaction can contain a parameter declaring it invalid until the block with a certain sequence number has been mined. This makes it possible to set an 'execution date' for transactions.

Note that out-of-order arrival may be the result of a number of factors: the forwarding behaviour of a node depends on the implementation and is different even between versions of the 'official' Bitcoin Core client. It may naturally also depend on the load on miners (leading to low throughput as evidenced by an ongoing community discussion[3]). Transient connectivity issues and Internet routing constellations may also be at play.[4] Also note that transactions may be rejected by the mempool for certain reasons. We explain these below as we encounter them.

[3] https://en.bitcoin.it/wiki/Block_size_limit_controversy.

[4] This is why projects such as Fibre (http://bitcoinfibre.org/public-network.html) aim to provide high-speed links between certain locations.

To observe transaction inclusion and commit times on Bitcoin, we ran an experiment twice to allow for varying network conditions and growth of the network. Each experiment lasted ca. 25 h; the first was conducted in November 2016, the second in April 2017. We collected roughly 300,000 transactions in each experiment. It should be noted that websites like https://blockchain.info/unconfirmed-transactions reported high network load while the second experiment was being carried out, with 25,000–30,000 transactions waiting for inclusion.

We summarize the commit times (using 6-confirmation) we determined in Table 11.1. Note that they are significantly higher and more varied in the second experiment. Figure 11.2 plots the commit times for the two forms of transactions that are our primary interest. The blue curves refer to transactions that were a 'straight-accept', i.e. the parent transactions were known and the incoming transaction passed all mempool tests. The violet curves are the transactions that were orphans upon arrival.

Table 11.1 Summary of commit time distributions (in seconds) for orphans and straight-accepts during our experiments

Type	Min	Q1	Median	Mean	Q3	Max
Experiment 1						
Orphans	944	3096	4635	7582	8334	117,585
Accepts	676	2887	4234	5475	5901	150,123
Experiment 2						
Orphans	1293	4280	6337	34,912	51,352	174,516
Accepts	1165	3873	5364	18,417	19,286	171,566

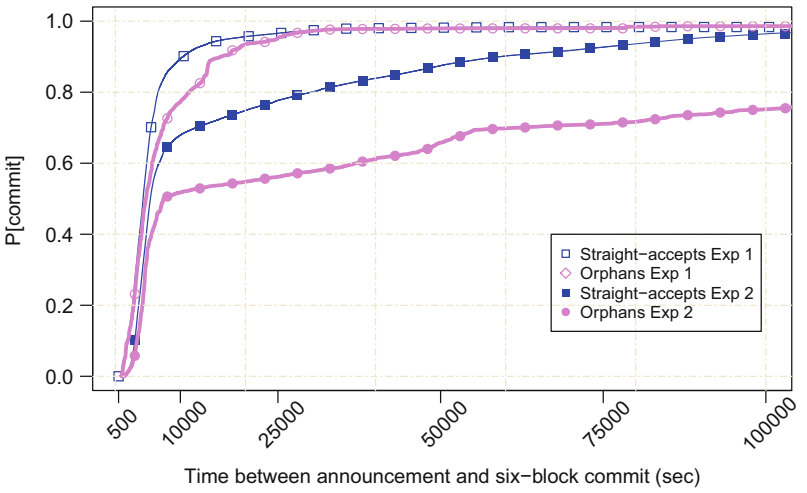


Fig. 11.2 Time between reception of transaction and commit. Note the logarithmic x-axis. © 2017 IEEE. Reprinted, with permission, from Weber et al. (2017)

In an underutilized network, the theoretical median 6-block commit time should be around 3900 s: six blocks of 10 min or 600 s, plus on average half an inter-block time of waiting time for mining on a new block to start. In experiment 1, the median commit time for straight-accepts was 4234 s and the 90th percentile 9501 s. For experiment 2, these times were 5364 s as a median and 55,976 s for the 90th percentile. In summary, *even if waiting twice as long as the median time, more than 10% of transactions were not committed yet.* This is an important factor to consider when building an application based on the Bitcoin blockchain: median commit times are high, but individual commit times can be much higher.

We then did a number of analyses to examine delays and orphans further. In both experiments, orphans seem to be committed later than transactions that were directly accepted. However, the additional delay is much higher in the second experiment (where the network was under high load). In our first experiment, about 60% of orphans were included within the same time span as normal transactions. In fact, 31% of orphans took longer than 2 h to be included, 21% longer than 3 h, and 8% took longer than 5 h. For directly accepted transactions, these values were slightly different: 17% of them took longer than 2 h, 9.5% longer than 3, and 5% longer than 5 h. In our second experiment, roughly 40% of orphans had similar commit times as directly accepted transactions. The majority experienced very significant delays: the median was almost 20% higher, and the third quantile is more than 2.5 times as high as that for straight-accepts. We also note that only 1.2% of orphans and 1.6% of directly accepted transactions had not been included by the end of our observation period in experiment 1. In experiment 2, more than 20% of orphans had not been included (but almost all straight-accept transactions).

Factors other than the out-of-order arrival might still exercise considerable influence on commit times. We hence decided to investigate two further factors: transaction fees and locktimes. We first determined the number of transactions with a zero fee. This was always very low: for the straight-accepts, it was 74 and 12 transactions in experiments 1 and 2, respectively. The orphans *never* had a zero transaction fee. Figure 11.3 shows a box plot of transactions fees with the zero values filtered out. We can see that transaction fees are considerably higher in the

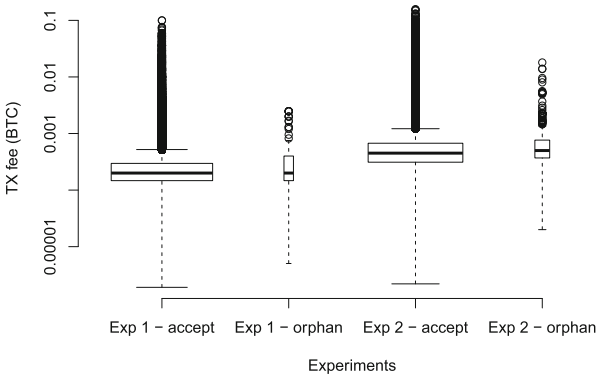


Fig. 11.3 Box plot of transaction fees by transaction category. Note the logarithmic y-axis. © 2017 IEEE. Reprinted, with permission, from Weber et al. (2017)

second experiment, but there is no difference between straight-accepts and orphans in experiment 1. In experiment 2, orphans even have slightly higher fees. It is very unlikely that lower transaction fees are a cause for delayed commit of orphans.

We extracted the locktimes for our collected transactions and the locktimes of their parents. As our logger had not captured the full content of transactions arriving in the mempool (but only hash value and timestamp), we conducted this analysis only for those transactions that had been incorporated into the blockchain. The vast majority of transactions had no locktime set: in experiment 1, only 15% of straight-accepts and 12% of orphans had a value that was not zero. In experiment 2, the numbers were 23% and 17%, respectively. While this may signal an increase in the use of the feature, orphans *never* had locktimes beyond the observation window. Orphans in experiment 1 had locktimes that ended at least 3 h before the end of the observation window; in experiment 2 it was 6 h. In contrast, straight-accepts did have locktimes that extended considerably beyond the end of the observation window. In experiments 1 and 2, nearly 100% of transactions also had locktimes similarly near the end of the observation window. However, we found some decidedly optimistic locktimes on the order of 1.5–1.7 billion (block sequence number). With 10 min being the average time between two Bitcoin blocks, these transactions cannot be included before the year 30,166. The obvious limitation of our work here is that we do not know the locktimes of those orphans that were not included in the blockchain by the end of our observation period. Given the above results, however, we still feel confident to say that locktimes are not likely to be a decisive factor in commit delay of orphans.

Naturally, there may still be confounding factors in our study that we could not control for in this experiment. For example, we do not have information about node connectivity outside of our observation post, Australia, and could not determine the (ever changing) Internet routing constellation that the Bitcoin network is exposed to. Note that propagation times in the Bitcoin network have been investigated before. Our study suggests that it is worthwhile to revisit this topic.

11.6.2 Variation in Ethereum Transaction Commit Time

In this section, we first explain why Ethereum transactions are not guaranteed to be committed regardless of their validity. We then analyse if gas price, gas limit, and the network as factors affect commit time.

Recall the life cycle of individual transactions in the Ethereum blockchain from Section 2.2, depicted in Fig. 11.4. It starts with the submission of a transaction into the (virtual distributed) transaction pool across all miners. A transaction lifespan can be split into consecutive phases: (i) the announcement of the transaction in the system; (ii) the inclusion of the transaction in a newly mined block on some branch of the chain; (iii) the inclusion of the transaction in a block part of the main chain; and (iv) the commit of the transaction after sufficiently many confirmation blocks are subsequently mined.

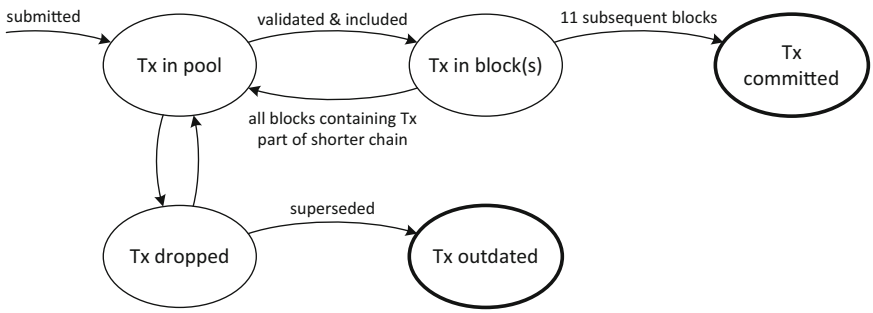


Fig. 11.4 Life cycle of an individual Ethereum transaction (notation: state machine; repetition of Fig. 2.7). © 2017 IEEE. Reprinted, with permission, from Weber et al. (2017)

There is no certainty whether a particular transaction will eventually be committed or whether it will be *outdated*, in that it will be considered an invalid transaction forever. Moreover, it is impossible to know whether a transaction that is invalid in some state of the system will never be valid in a later state. More specifically, the aforementioned step (ii) is not sufficient to guarantee that a transaction *Tx* is permanently added to the blockchain: if the blockchain forks, then the block comprising the transaction may simply be discarded, in which case the transaction could be re-included later.

To put it differently: there are only two final states in this life cycle, namely, *committed* or *outdated*, and only these and inclusion in a block are observable transaction states for each client. In order to build a robust application on this basis, one needs to ensure that each transaction ends up in one of the final two states in a reasonable time. Otherwise the status of the transaction is, from the viewpoint of the client, undefined and unknown.

When a transaction is included in a block, it has been validated beforehand, i.e. its digital signature has been checked, as well as the validity of parameters like the nonce (sequence number of transactions relative to a given source account), and that there are sufficient funds in the source account. If all blocks that included the transaction become *uncles*—i.e. part of a shorter chain than the main chain—then the transaction goes back into the transaction pool. This may happen more than once, and, theoretically, there is no upper limit. While the transaction is in the pool, it may also be dropped. This is a local decision of miners, and it is impossible for any node in the network to know with certainty that all miners have dropped the transaction. Only when the nonce of the transaction becomes outdated, i.e. another transaction from the same source account with the same nonce got committed, can a node be certain that the transaction is invalid and will not be included in any valid block. Otherwise the transaction may resurface at a later point and get included in the chain.

Ethereum's transaction handling and inter-block time differ significantly from Bitcoin, and the chance of a chain fork occurring is higher. If a fork occurs, there is usually no certainty as to which branch will be permanently kept in the blockchain

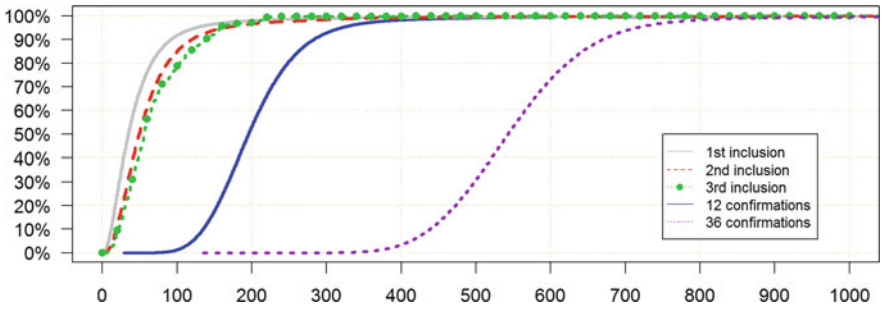


Fig. 11.5 Time (s) for first inclusion and commit (12 or 36 confirmations), as well as second and third inclusions of transactions that were previously included in uncles. © 2017 IEEE. Reprinted, with permission, from Weber et al. (2017)

and which branch(es) will be discarded. In particular, transactions that were only included in uncles need to go back to the transaction pool. Before investigating the factors that cause commit delays, we investigate how fast transactions proceed from first inclusion to commit.

To empirically investigate transaction inclusion time in Ethereum, we collected data on *approx.* 6 million transactions over a 3.5-month period, discarding any short periods affected by network or system outages. The observations were conducted between December 2016 and April 2017. Figure 11.5 depicts the observed distributions of the time it takes for an Ethereum transaction to be included in a block and committed (using 12-confirmation, i.e. 11 subsequent confirmation blocks after inclusion, and 36-confirmation).

As shown in the figure, the inclusion times tend to follow similar curves. However, compare the slopes of the curves for first to third inclusion to the slopes for 12-confirmation and 36-confirmation: the latter are less steep, indicating the growing fraction of transactions that have to wait longer for a 'commit'. For a '12-block commit', the median time is around 200 s, and even the third quantile is not much higher. But the more blocks we require for a commit (say, 24 or 36 blocks), the more likely it becomes that a transaction needs (even considerably) longer than the median would suggest.

In contrast to Bitcoin, for the observed period the 90th percentile of commit happened significantly earlier than twice the median: at about 270 s for 12 blocks and around 650 s for 36 blocks. Still, while the curves converge towards 100%, they do not reach it within 1000 s. As a consequence, *applications sending larger volumes of transactions need to be prepared that some of these will not be committed in due time.*

Concerning transactions that become 'unincluded', however, we find that these are rare indeed. We observed that 113,122 first transaction inclusions (0.021%) were not permanent; and the same is true for 2602 second inclusions (0.0005%) and 41 of the third inclusions (0.000007%).

Ethereum has two user-defined parameters around the concept of gas, namely, the gas price and the maximum gas offered for including a given transaction. We

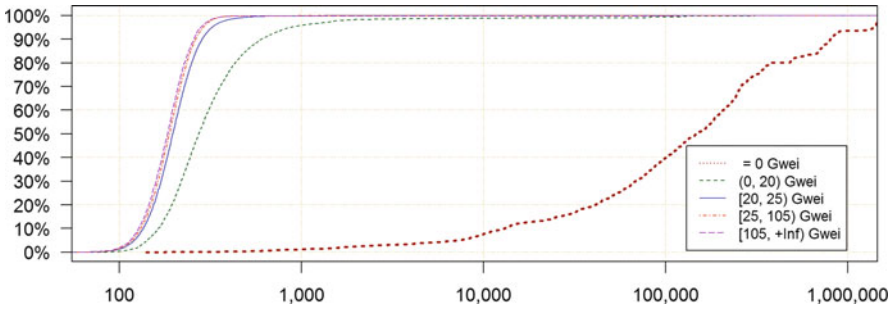


Fig. 11.6 Commit delay (s) for transactions based on gas price. Note the logarithmic x-axis. © 2017 IEEE. Reprinted, with permission, from Weber et al. (2017)

proceeded to investigate how these affect the commit times. In particular, we were interested to see if it is possible to speed up the commit time by offering particularly high rewards for miners by setting a high gas price.

Based on our collected data, we analysed the effect of the user-defined gas price on the time it took for the transaction to be committed. Figure 11.6 depicts this relation for five bands of gas price (all in Gwei[5]): $[0, 0]$, $(0, 20)$, $[20, 25)$, $[25, 105)$, $[105, +\infty)$.

As shown in the graph, the higher the gas price in a given band, the less likely we observed long delays. However, we did not observe any meaningful differences from 25 Gwei onwards. At the time of writing in 2018, this observation is unlikely to hold true to the same degree: from anecdotal evidence, it appears miners behave more rationally. Finally, there is a sharp contrast between the 0-band and all other bands: the 0-band has significantly longer commit times.

A second user-defined variable around transaction fees is *maximum gas*, i.e. how much gas the execution of the transaction may use. We analysed its impact on commit delay. While we discovered individual transactions that were delayed due to an exceedingly high gas limit, our analysis was inconclusive: we could not find a strong correlation in any direction between maximum gas and commit delay. This remains an open question for now and warrants longer observation.

We were also curious to see whether the Ethereum network suffered from transaction reordering as we had observed it for Bitcoin. Ethereum does not link transactions in the way Bitcoin does, but every transaction has a running sequence number ('nonce') for each sender account. This sequence number starts from 0 and increments by 1 for each transaction sent from the same account. It is intended to provide an assurance that transactions from the same account will be executed in a particular deterministic order. However, it also means that a transaction with a nonce $n + 1$ cannot be included into the blockchain unless there is an already included transaction with nonce n—it is 'orphaned'. The transaction with the higher nonce will wait in the transaction pool until the arrival of a transaction with n as nonce.

[5] 1 Ether are 10^{18} wei.

We hence carried out an experiment that is similar in nature to our previous Bitcoin experiment. We analysed the commit times for *in-order* and *out-of-order* arrival of transactions during the same interval as for our second Bitcoin experiment, in April 2017. The total number of transaction announcements, which were also committed during this period, was 87,384. The number of transactions with out-of-order nonces was 5403 (6.18%). The commit time for both categories is shown in Fig. 11.7. The graph suggests that the commit delay for *out-of-order* transactions is almost doubled, compared to *in-order* transactions. To exclude the gas price as a confounding factor, we plot the gas price distribution for both categories, shown in Fig. 11.8. We did not find a significant difference in gas prices between two categories.

As with Bitcoin, it is hard to rule out other confounding factors that we cannot control for, e.g. Internet routing or overall network connectivity. However, our data allowed us some partial insight into the latter. We inspected transactions with nonce n that were announced *after* transactions with nonce $n + 1$ and compared these with *in-order* transaction announcements. Figure 11.9 plots the distribution of unique Ethereum nodes that we saw broadcasting the transaction before inclusion in the block. We found that delayed transactions were known to much fewer nodes. While

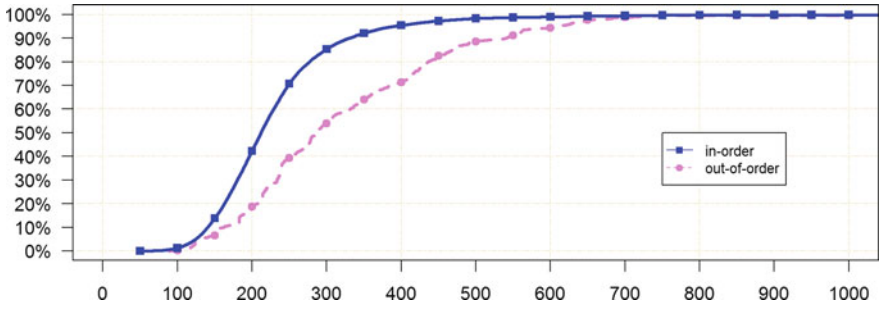


Fig. 11.7 Commit delay (s) for transactions based on ordering. © 2017 IEEE. Reprinted, with permission, from Weber et al. (2017)

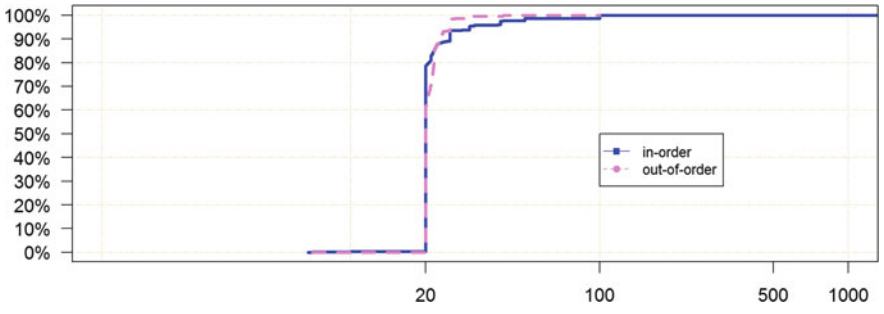


Fig. 11.8 Gas price distribution (GWei) for transactions based on ordering. Note the logarithmic x-axis. © 2017 IEEE. Reprinted, with permission, from Weber et al. (2017)

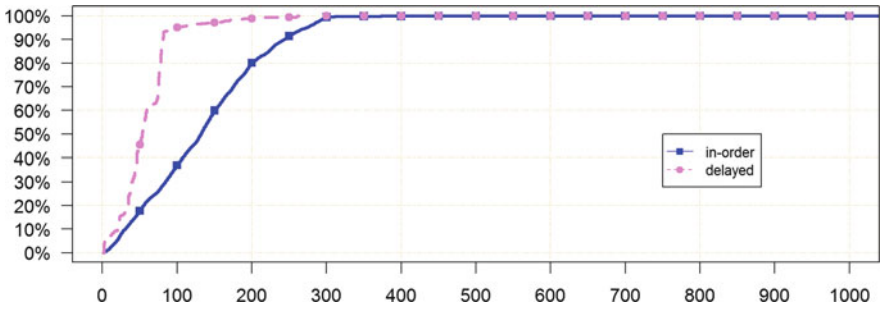


Fig. 11.9 Number of different peers from which in-order and out-of-order transactions arrive. © 2017 IEEE. Reprinted, with permission, from Weber et al. (2017)

not conclusive, this provides first indications that network connectivity may have negatively impacted transaction propagation.

Ethereum has a second form of limit, the so-called gas limit per block. Unlike the gas price in a transaction, it is defined by the network of miners and applies to the *sum* of gas consumed by all transactions in a block. If the limit is lower than the gas required for a given transaction, the transaction cannot possibly be included. The development of the gas limit over time is readily available, e.g. on Etherscan.[6]

The rationale for the limit is to prevent denial-of-service (DoS) attacks on the network by limiting the amount of computation that can be done per block. Due to several DoS attacks against the network, a majority of miners on Ethereum agreed to lower the limit to *approx.* 500,000 gas temporarily—from October 15 to 17, 2016, according to Etherscan. The network still kept a low limit prior to and after these 3 days: from September 23, 2016, to November 22, 2016; with 1-day exception, the limit was around 2M gas. Around December 5, it returned to 4M gas. This limitation can negatively impact the inclusion of transactions which require a high amount of gas. This is not a hypothetical case: in earlier work, we deployed contracts using around 1.5M gas ourselves. However, simple transfers of assets should not be negatively impacted.

We hence chose to investigate whether we could find evidence for this hypothesis in our data. We analysed all transactions that happened before the DoS attacks and used block 2,303,121 as the pre-DoS cut-off block. We considered the amount of gas used for three different types of transactions: financial transfers, regular function calls to contracts, and contract creation.

Figure 11.10 shows the distribution of gas used for these transaction types. It highlights the gas limits mentioned above as vertical lines. No *financial transfer* transaction used more than 100,000 gas. This was an expected finding, as a financial transfer will incur 21,000 gas as base cost for any transaction, plus possibly a small amount for attached data: between 4 and 68 gas per byte (used, e.g. for a description

[6] https://etherscan.io/chart/gaslimit.

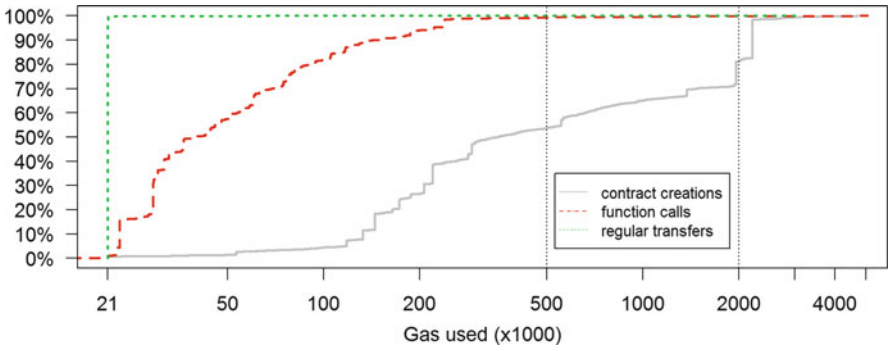


Fig. 11.10 Distribution of gas usage for different types of transactions, prior to DoS attacks. Dotted vertical lines show limits in response to the attacks. © 2017 IEEE. Reprinted, with permission, from Weber et al. (2017)

of the transfer. As for *function call* transactions, 94% of them used at most 200,000 gas. Only 0.62% of the remaining function call transactions would not have been possible with the 500,000 gas limit. This contradicted a part of our hypothesis and highlighted that most of the functions that were in use were not highly demanding in terms of computation or storage.

However, when inspecting contract creation, we found that only 53.79% of all the contracts created before the DoS attack could have been created with the 500,000 gas limit, while 46.21% required more gas. This confirmed our hypothesis that many contracts would not have been deployable while the low block gas limit was in place. Even for the 2-month period where the network kept the block gas limit at about 2 million, 18.78% of contract creation transactions would have been impossible.

11.7 Aborting and Retrying Blockchain Transactions

One issue for a system designer who is building a blockchain-based system is that there is *no option to abort* a transaction. In this section, we propose a mechanism to artificially abort Ethereum transactions by superseding them with an idempotent or counteracting transaction. This abort mechanism can be useful if, for instance, the system observes that the transaction has not been committed within a specified time frame (as can be the case with, e.g. orphans). As such, the retry and abort mechanisms could be implemented to increase the user-friendliness or robustness of software clients or wallets.

Another motivation for abort is the accidental duplication of transactions, which we discovered thousands of times in our observation of Ethereum: the same transaction was submitted twice, often within seconds, but with different nonces, and the funds in the sender's account were insufficient for both transactions to execute. Seemingly the senders thought they were retrying the same transaction,

when really they created two separate transactions with the same parameters, except for the nonce. Instead of transferring the desired amount once, it would be transferred twice (if the balance was or became sufficient). This has also happened to individuals we know personally.

11.7.1 Aborting and Retrying Transactions in Ethereum

There are some options to achieve an effect that is similar to an explicit abort. In Ethereum, for instance, the system or user can issue a competing transaction from the same source account, i.e. another transaction with the same nonce. Assume user Alice transfers 1 Ether to Bob by issuing transaction Tx_i with nonce i. After an acceptable time frame, e.g. 10 min, has elapsed and Tx_i has not been committed, Alice wants to abort Tx_i. She then submits a new transaction Tx_i', with the same nonce i as specified in Tx_i and a higher transaction fee in order to increase the chances for Tx_i' to be included. For this transaction Tx_i', she does not want to spend more Ether than necessary; thus, she sets the transaction value to 0 and her own account as receiver. Once Tx_i' is committed, Tx_i is superseded by it and becomes outdated. If, in the meantime, Tx_i were to succeed, Tx_i' becomes outdated. This is acceptable, since that was the original intent.

Alternatively to aborting, Alice can 'retry' Tx_i by submitting Tx_i'' as follows: the fields in Tx_i'' contain the same data as in Tx_i, including nonce i—except Alice offers a higher fee for it. Therefore, the hash and digital signature of Tx_i'' will be different from Tx_i, and thus it will be perceived by the miners as a separate transaction. If Alice tried resending Tx_i without any changes, hash and signature would be the same, and the miners would not consider it any differently—unless they have previously dropped Tx_i. In the latter case, the reasons for dropping Tx_i might not have changed, and thus the same would likely happen again. If either Tx_i or Tx_i'' succeeds, the respective other transaction would become outdated and invalid, since they both have the same nonce i.

11.7.2 Aborting and Retrying Transactions in Bitcoin

The Bitcoin blockchain does not offer transaction abort. In a German newspaper article from late 2017, the author described that he 'lost' BTC[7] worth several hundred Euros, since he did not offer a transaction fee, and his transaction had not been included for more than 2 weeks. His wallet application did not offer options to abort or retry the application, and simply reported his account balance to be zero. In cases like that, we believe the following method should work.

[7] BTC is the currency code for Bitcoin's cryptocurrency.

Say, Alice wants to transfer 5 BTC to Bob. She previously received 2 BTC from Charlie, as output 0 (abbreviated as #0) in Tx_1, 1 BTC from David as #0 in Tx_2, and 4 BTC from Erin as #1 in Tx_3. Her virtual account thus holds 7 BTC. To achieve the transfer, Alice creates transaction Tx_{orig} that has two inputs: Tx_1 #0 (2 BTC) and Tx_3 #1 (4 BTC), so that Tx_{orig} has a transaction volume of 6 BTC. Alice then adds two outputs: #0 with 5 BTC to Bob and #1 with 0.99 BTC to herself. Tx_{orig} thus offers a transaction fee of 0.01 BTC, and subsequently her virtual account will hold 1.99 BTC.

Now, say the commit of this transaction does not happen within Alice's timeframe of 6 h and Alice wants to abort. Since each input can only be spent once, Alice can achieve that by submitting Tx_{abort} with the same inputs as Tx_{orig}, but as single output #0 she specifies 5.98 BTC to herself (thus offering a transaction fee of 0.02 BTC). If either Tx_{orig} or Tx_{abort} succeeds, Alice's account is not in limbo, and she can continue to use the network as normal.

As an alternative to abort, Alice can re-attempt the transfer with Tx_{retry} as follows. The inputs are the same as in Tx_{orig}, output #0 stays at 5 BTC to Bob, but output #1 is changed to transfer 0.98 BTC to herself. Tx_{retry} thus offers a higher transaction fee of 0.02 BTC, and if Tx_{retry} succeeds then Tx_{orig} becomes outdated.

11.7.3 Experiments for Aborting Transactions in Ethereum

We tested the above method for abort on the public Ethereum blockchain for three scenarios: (i) a transaction does not get included in the usual period of time; (ii) a client changes its mind and decides to roll-back the issued transaction; and (iii) a transaction is in indefinite pending state due to insufficient funds. We describe these below in more detail.

Abort Experiment 1 In order to test the situation where a sent transaction does not get included in the usual timeframe, we submitted 100 transactions that *underbid* the market rate. Specifically, we assumed the average gas price from the previous day (December 1, 2016) as market rate (mr) and submitted ten transactions each for different prices, which are 0, $0.1 \times mr, 0.2 \times mr, \ldots, 0.9 \times mr$. As cut-off time, we rounded up the 99% percentile from our earlier experiment (cf. Fig. 11.7) to 10 min. If the transaction had not been included then, we submitted an abort transaction Tx_{abort} as described above, with the same nonce but at full market rate mr, target `0x0`, and value of 0.

The results are shown in Fig. 11.11. Surprisingly, most transactions were accepted by the network. Six out of ten transactions with either 0 or $0.2 \times mr$ were accepted. In addition, only two out of ten transactions with $0.1 \times mr$ were accepted. All of the 16 timed-out transactions were successfully aborted with our Tx_{abort} mechanism described above.

Abort Experiment 2 For this experiment, we assumed a client that underbids the market fee and changes its mind regarding an issued transaction. As in the previous

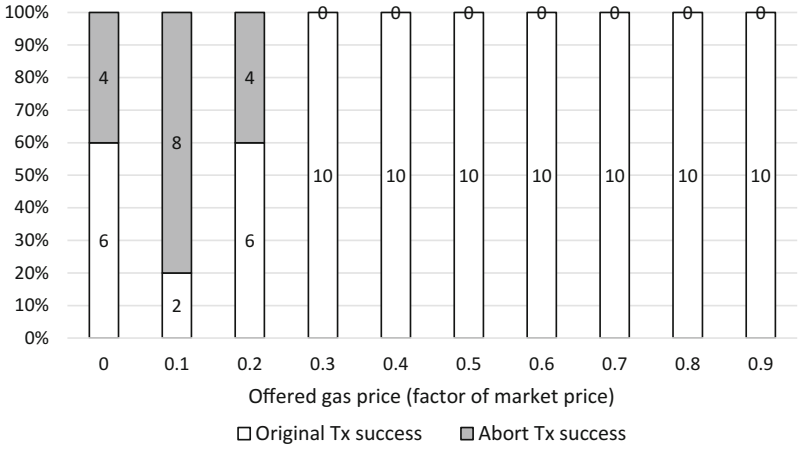


Fig. 11.11 Underbidding market fee and automatic abort after 10 min if the original *Tx* was not included. © 2017 IEEE. Reprinted, with permission, from Weber et al. (2017)

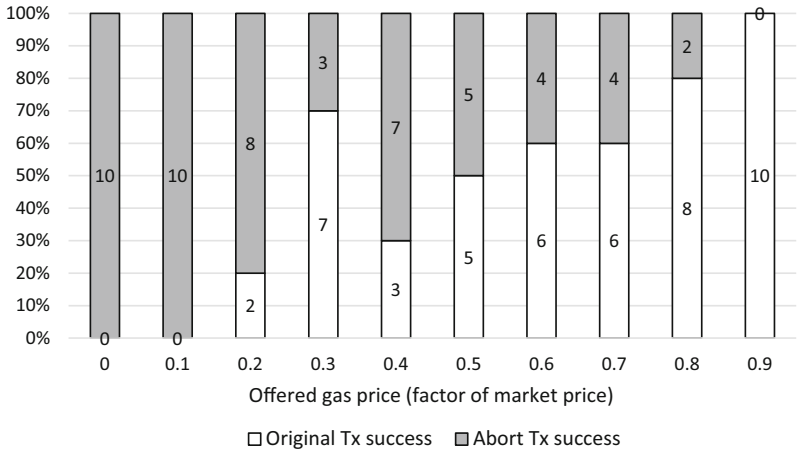


Fig. 11.12 Underbidding market fee and automatic abort after 3 min if the original *Tx* was not included. © 2017 IEEE. Reprinted, with permission, from Weber et al. (2017)

experiment, we sent another 100 transactions with gas prices as above, i.e. 0, $0.1 \times mr$, $0.2 \times mr$, ..., $0.9 \times mr$ for ten transactions each. Rather than waiting for 10 min, we set the timeout value to the target median for Ethereum transaction commit, i.e. 3 min.

The results of this experiment are shown in Fig. 11.12. A much higher percentage of transactions were not included in a block after 3 min, in comparison to Fig. 11.11 with 10-min timeout. As before, 100% of Tx_{abort} succeeded. Interestingly, all of them were included in a block after 3 min. In 2 out of the 100 cases, the 3-min timeout for the original transaction was reached, Tx_{abort} was sent, but the original transaction Tx_{orig} still won the race and got included and committed in

the blockchain. Thereby, Tx_{abort} was outdated. As stated above, this is a possibility that clients should be prepared for. The reasons for such a situation can include (i) processing time in our client when preparing the Tx_{abort}; (ii) broadcast delays or other network effects where the winning miner does not receive Tx_{abort} before including Tx_{orig}; or (iii) non-rational scheduling of transactions in the pool, where no preference is given to the transaction with the higher fee.

Abort Experiment 3 In this last experiment, we submitted two transactions, creating a situation that corresponds to faulty inputs from a user (or user's program). We have observed such behaviour during our live observation of the public Ethereum blockchain. To replicate it, we submitted two transactions, Tx_1 and Tx_2, as follows. Assume that the last nonce for the sender address was n and its account balance k. Then we create Tx_1 with nonce $n + 1$ and value $\frac{1}{1000}k$ and Tx_2 with nonce $n + 2$ and value $\frac{999}{1000}k$. For both transactions, we set the gas price to $0.7 \times mr$. Due to the nonce, Tx_1 must be included before Tx_2. However, due to the positive gas price, the account balance resulting from the inclusion of Tx_1 is insufficient for Tx_2.

Finally, we submit Tx_2, wait 5 s, and then submit Tx_1. This gives Tx_2 the chance to get broadcast before Tx_1 is known to any node, including our own. This procedure is needed so that the client submits Tx_2 to the network; since geth is not aware of Tx_1 and its contents when we submit Tx_2, it broadcasts Tx_2. Otherwise, it might detect the insufficient balance and not accept Tx_2.

Once Tx_1 has been included in a block, Tx_2 is invalid due to insufficient funds. However, this does not always get checked, and hence Tx_2 may remain in the transaction pool for a long time. In fact, if another transaction deposited funds into the sender account, Tx_2 would become valid and be executed. This, again, is behaviour that we observed. Here, we send a Tx_{abort} with the nonce $n + 2$, to abort Tx_2.

We ran this experiment until we had submitted Tx_{abort} 100 times. All 100 submitted Tx_{abort} were successful. We measured the time it took for Tx_{abort} to be included in a block (first inclusion) and plotted that as shown in Fig. 11.13. The median for those times was 45 s and the maximum 230 s.

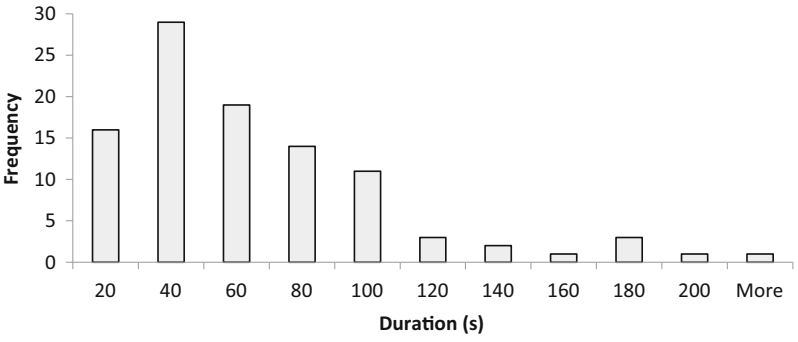


Fig. 11.13 Abort duration histogram, from experiment 3. © 2017 IEEE. Reprinted, with permission, from Weber et al. (2017)

Our experiments support the hypothesis that transactions can be aborted with our proposed method. Although it would be better to have explicit abort mechanisms for blockchains, this is a fall-back method for certain applications to address commit delays that are due to some of the factors we have described in Section 11.6.2.

11.8 Summary

We started this chapter with a broad discussion on the impact that using blockchain as a component can have on dependability and security properties. In short, confidentiality can be harder to achieve, due to the replication of the data structure to the whole network; integrity is blockchain's strong suit; in terms of safety, the picture is less clear; maintainability requires planning and governance; and availability and reliability features are high for reading/receiving, but potentially low for writing/sending.

To give a clearer picture of the write/send availability and reliability characteristics, we studied the public Bitcoin and Ethereum networks. For Bitcoin, we found that even if waiting for a transaction commit twice as long as the median time, more than 10% of transactions were not committed yet. For Ethereum, this was less common, but still above 1%. This is important when building an application based on public blockchains: commit times vary significantly and can take significantly longer than in common cases.

Finally, we discussed methods for transaction abort and retry, which are not built-in functions of blockchain clients. Applications can use these methods to handle transactions that take unusually long.

11.9 Further Reading

This chapter is partly based on Weber et al. (2017) and draws on earlier ideas from Anderson et al. (2016).

As stated in the beginning of the chapter, we did not cover security infrastructure or cryptography. A number of books discuss these points in detail, e.g. Bashir (2018).

In this chapter, we refer to a few seminal works, specifically the Clark–Wilson security policy model (Clark and Wilson 1987), and the taxonomy of dependable and secure computing by Avizienis et al. (2014). The alternative definitions of safety are described in Lamport (1977) and Alpern and Schneider (1985). Finally, the CAP theorem (Fox and Brewer 1999) indicates that there is inevitably some trade-off between consistency, availability, and partition-tolerance for distributed databases.

The Ethereum yellow paper (Wood 2015–2018) specifies gas costs for various operations and describes the function of block gas limits.

As mentioned in the previous chapter, live statistics about the public Ethereum chain, including inter-block times and the influence of the gas price on transaction inclusion times, are available at https://ethstats.net/ and the ETH Gas Station (https://ethgasstation.info/). ETH Gas Station also gives recommendations for gas price settings, relative to desired inclusion times; these can also be accessed through an API. From these recommendations it appears that, at the time of writing, miners now react more to gas prices and the network is less likely to accept transactions offering no fee than it did when we conducted our experiments.

An earlier investigation on propagation times in the Bitcoin network has been conducted by Decker and Wattenhofer (2013).

Part IV
Case Studies

Chapter 12
Case Study: AgriDigital

Blockchain Technology in the Trade and Finance of Agriculture Supply Chains

with Bridie Ohlsson and Katherine Davison

12.1 Agricultural Supply Chains

12.1.1 Global Agricultural Supply Chains

Agriculture is an industry that provides food and fibre to feed and clothe the world's 7 billion citizens. Over 25% of the world's working population are employed in the agriculture sector.[1] In order to meet demand in a globalized world, agriculture supply chains have formed as long and complex networks of production, processing, distribution, and marketing channels. They are made up of farmers, processors, traders, logistics providers, financiers, consumers, and many others, each having widely varied and often competing interests.

Globally, agriculture is a $6 trillion industry.[2] Agriculture broadly includes the production of commodities such as rice, corn, wheat, livestock, cotton, and vegetables. While each individual supply chain is unique, across many geographies and commodities agricultural supply chains face common challenges.

Counterfeit Goods Supply chain participants are vulnerable to fraud, with global food fraud costing US$40 billion annually.[3] More broadly, the annual global trade in fake goods amounts to a staggering US$460 billion.[4] Without verifiable and data-rich physical assets, counterfeit goods move in large quantities along supply chains. The human cost of food security has become a very real challenge in many parts of the world.

[1] International Labour Organization, ILOSTAT database, 2017.

[2] World Bank and OECD National Accounts data, 2016.

[3] John Spink, Michigan State University Food Fraud Initiative 2014.

[4] OECD & EUIPO 'Trade in Counterfeit and Pirated Gods: Mapping Economic Impact', 2016.

X. Xu et al., *Architecture for Blockchain Applications*,
https://doi.org/10.1007/978-3-030-03035-3_12

Counterparty Risk Payment security and counterparty risk are faced by buyers across the supply chain. Often, buyers and sellers cannot operate with confidence, because they do not know that they will receive timely payment for their commodities and be able to access the finance necessary for business stability and growth. Farmers often do not receive payment for their products and commodities until months after delivery.

A lack of liquidity across the supply chain makes access to finance a real challenge for buyers, who are unable to pay farmers in a timely way. Commodity finance is often limited to reputable borrowers with bricks and mortar security, and is often only accessible for commodities where the risk price can be hedged. This results in settlement latency, with title transferring months before payment is made, introducing enormous counterparty risk which most often falls on the producer at the start of the supply chain. A key challenge faced by the bulk commodity supply chains has been providing clear visibility over commodity ownership. Paper-based systems or spreadsheets provide little to no security for farmers when payment fails.

Cooperation Supply chains are typically characterized by competition rather than cooperation. Individuals and organizations along supply chains lack the trust and incentive to openly share data around the status of goods. Only 43% of agri-supply chain participants feel confident that they can collaborate with their counterparties.[5]

12.1.2 Blockchain and Agriculture

Despite the overall digitization of the global economy, agriculture remains one of the world's least digitized industries.[6] Agriculture missed out on many of the benefits of the 'first wave' of the internet and associated technologies, due to a lack of connectivity and ready technical skills. Global trade is still largely paper-based and manual, and information around a commodity does not flow freely between supply chain participants. Costly back-office reconciliation processes, and manual double data entry, continue to add additional costs and human error into agri-supply chains.

Supply chains are consistently recognized as a natural market for blockchain technology. These are networks where multiple participants operate who do not trust each other but who require access to a single set of verifiable data and claims about a common asset. The natural state of agri-supply chains is distributed networks relying on a single source of truth. This led AgriDigital to see blockchains as critical components for building robust digital supply chains.

[5] 3M, Supplier Survey Whitepaper, 'Driving Growth and Innovation through Supplier Partnerships', 2017.

[6] McKinsey Global Institute Industry Digitization Index 2015.

12.2 The AgriDigital Vision

12.2.1 Building Digital Trust

Since being founded in 2015, AgriDigital has been using blockchain technology as part of a technical stack to build digital trust across agriculture supply chains. At the core of the AgriDigital vision is a platform and community approach to digitizing agriculture, and that doing so will bring security, trust, and value to agri-supply chains.

For AgriDigital, *digital trust* means supply chain participants can transfer commodities with complete security, can accurately attribute value to those goods, and can recognize financially and otherwise where that value has been contributed along the supply chain. The building of digital trust is the accumulation of a robust digital infrastructure, comprising multiple different components. Blockchain is part of the technical solution that will deliver digital trust to agriculture supply chains globally (Fig. 12.1).

AgriDigital was founded by a team of Australian farmers and agribusiness professionals. The AgriDigital founding team has a combined 80 years' experience in the grains industry and deep personal experience of the challenges agriculture faces.

Taking the approach of first delivering a cloud-based platform to market in the Australian grains industry, AgriDigital has been able to gain commercial traction

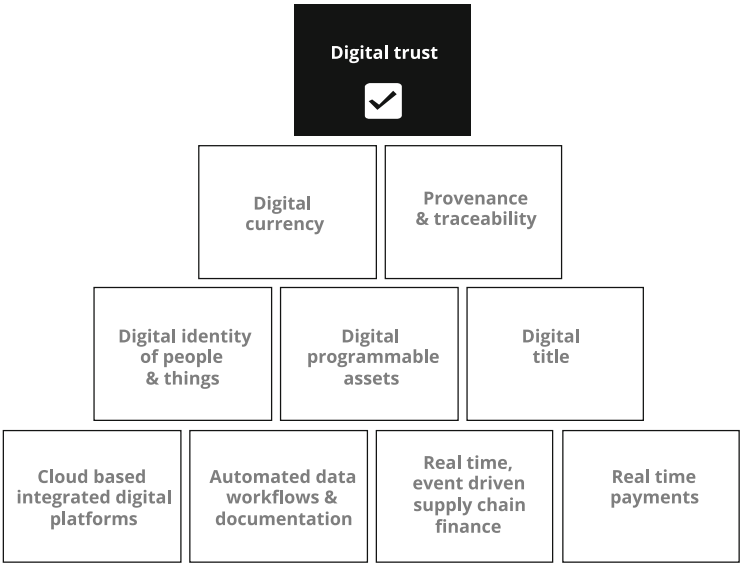


Fig. 12.1 Building blocks of digital trust in AgriDigital's vision. © 2018 AgriDigital, reprinted with permission

while continuing to pilot and test components of blockchain and other technologies. The AgriDigital platform is a commodity management solution connecting farmers, traders, and site operators to seamlessly manage contracts, deliveries, inventory, invoices, and payments. Through the application layer, AgriDigital is able to capture and validate information about the physical commodity and streamline interactions between supply chain participants.

The AgriDigital platform acts as the application layer through which customers can leverage the benefits of blockchain technology across agri-supply chains. Using this interface, AgriDigital has conducted world-first proof of concepts and pilots with leading agribusinesses, applying blockchain technologies to help solve embedded agri-supply chain challenges.

12.2.2 AgriDigital's Blockchain Solution

AgriDigital has designed a library of smart contracts to facilitate the trade and finance of agricultural commodities. In traditional supply chains, trade, finance, and data flows are kept separate, both within organizations and between them. This siloed approach to data flows contributes to the risks, delays, and fraud across agri-supply chains, as participants have a difficult time verifying commodities, managing costly processes to do so, and only trusting a limited number of counterparties (Fig. 12.2).

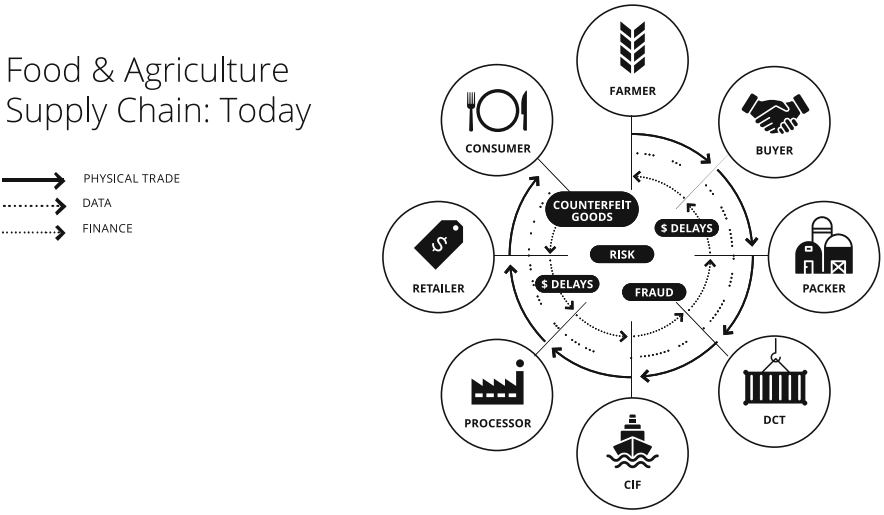


Fig. 12.2 Traditional agri-supply chains, with separate flows for trade, data, and finance. © 2018 AgriDigital, reprinted with permission

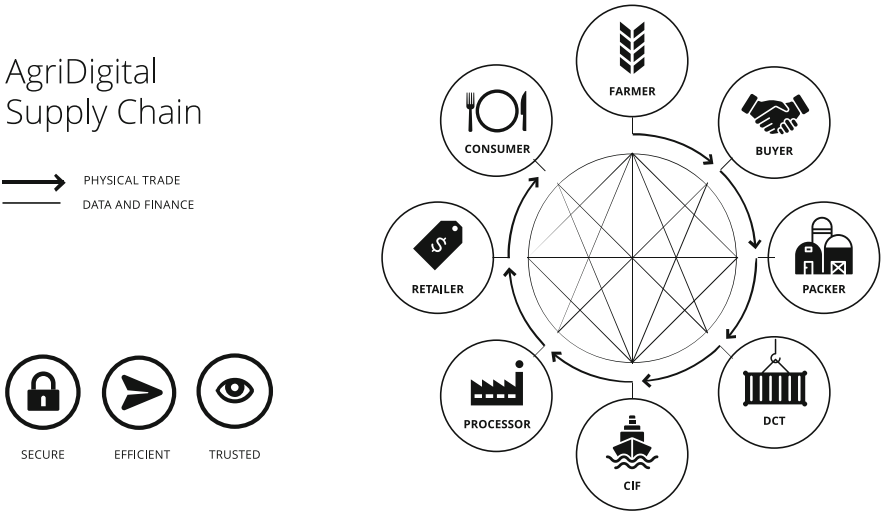


Fig. 12.3 AgriDigital solution: integrating the flow of data, trade, and finance using digital technologies. © 2018 AgriDigital, reprinted with permission

There is an enormous opportunity to drive value and innovation along supply chains, by bringing together these otherwise disparate information flows. Not only would this provide network and market efficiencies, but it would also act as a single source of truth providing supply chain assurance and transaction security. Bringing together trade, finance and data flows, applications, and individual users in one solution can leverage that information and contribute to solving challenges such as the lack of liquidity, counterparty risk, and counterfeit goods (Fig. 12.3).

At the core of the AgriDigital solution is the creation of digital assets. The digital assets become the anchor from which these trade, finance, and data flows are brought together. As the digital asset moves from participant to participant along the supply chain, an immutable and data-rich record of the physical asset is created on the blockchain-based protocol.

Once a digital asset is issued, participants can attach data including certificates and production records, seamlessly sharing this verifiable information with others in the network. Using smart contracts, counterparties can execute secure transactions, providing transparent chains of custody and proof of ownership. With full visibility over the asset, financiers can provide supply chain finance in new and innovative ways.

12.2.3 Architecturally Significant Non-functional Requirements

The vision and business context described above drive several architecturally significant non-functional requirements for AgriDigital's blockchain solution. The key quality is *integrity*, to ensure secure records of title and confidence in payments. *Transparency* of blockchain data to participants and *security* of transactions build digital trust in the physical asset. However in negotiating the commercial use case, complete transparency may only exist between specified parties; for other parties and especially competitors, there may be a need for data *privacy*.

The need for real-time payments and the need to support transactions in the context of deliveries in the physical world also drive requirements for acceptably low *latency*. Implicit in the vision for an industry-scale platform is a demand for high *availability* and acceptably high *throughput scalability*. All of these qualities are likely to be also requirements for other blockchain-based supply chain platforms. An interesting non-functional requirement directly related to AgriDigital's technology strategy is system *adaptability*, to allow the trial and gradual digitization at the application layer to incrementally increase the data richness of digital assets within the blockchain-based platform.

12.2.4 Pilots and Proof-of-Concept Overview

AgriDigital has conducted a number of proofs of concepts with industry participants. We give an overview of three of them here, before delving into more in-depth details of the second proof of concept in the next section.

Pilot 1: Fletcher International Exports
December 2016, Dubbo, NSW

In December 2016, AgriDigital executed the world's first settlement of a physical commodity on a blockchain. This pilot was significant in its delivery of real-time settlement for physical commodity transactions, opening the doors for elimination of the counterparty risk that all sellers face.

An Australian wheat farmer, David Whillock from Geurie, NSW, delivered 23.46 metric tonnes of wheat to Fletcher International Exports (FIE) who run a processing and exports business in Dubbo, New South Wales. In a global first, Whillock was paid instantaneously using blockchain technology.

A smart contract from the proprietary AgriDigital library auto-executed the settlement. At the moment of delivery to the silo at Dubbo, the quality and quantity of wheat being delivered was recorded at the weighbridge and sample station. The smart contract then valued the particular delivery of wheat against an existing legal contract, then verifying FIE had sufficient funds in their digital wallet to pay Whillock and securing the funds in Whillock's name pending delivery

confirmation. Once the farmer completed the physical delivery at site, title to the grain transferred from Whillock to FIE, and simultaneously payment was made from FIE to Whillock.

For this pilot, though transaction settlement occurred on the blockchain, Whillock received the payment in local currency using traditional banking methods: a message was sent out as a bank file for the buyer to upload and pay, on the same day, via existing payment mechanisms. Typically, payment terms in the Australian grains industry range from 2 to 5 weeks, and these terms are the ones that pose counterparty or credit risk to growers. Using smart contracts to match title transfer to payment provides instant benefits to farmers and other sellers by removing counterparty risk and increasing security over the asset up until the moment the title transfers.

The pilot ran in December 2016 using a private instance of the Ethereum blockchain. AgriDigital managed the three-node network, simulating the situation where AgriDigital, the buyer, and a third-party regulator each operated a full blockchain node. The private blockchain was configured to mine approximately one block per second, where each block may or may not contain transactions. The delivery information was provided through integrations with electronic weighbridges, which automatically created messages as measurements were taken and sent these to the AgriDigital system. Additional manual data entry from the sampling station was entered into the AgriDigital frontend. At the time, Ethereum was limited in its handling of decimal values, and thus some rounding error occurred as expected.

Pilot 2: CBH Group
July 2017, Bordertown, SA

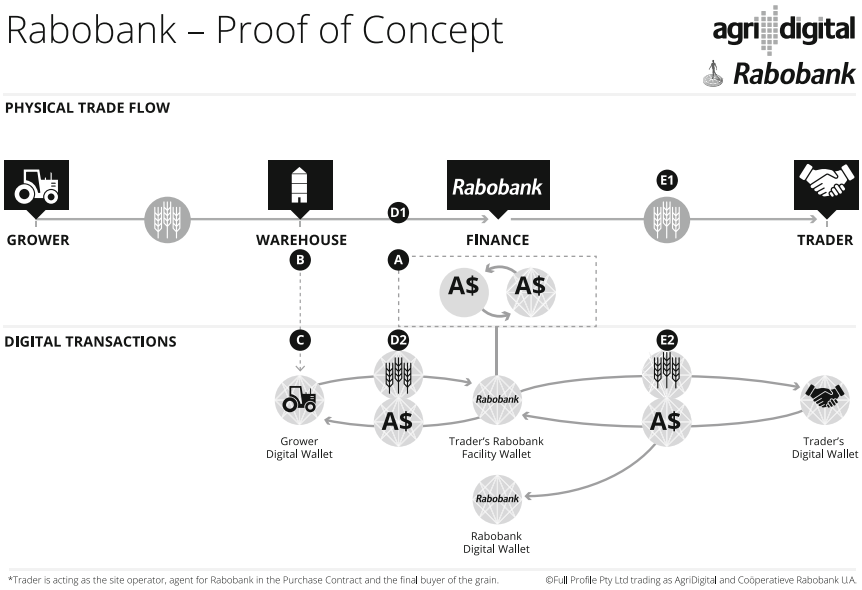
In partnership with CBH Group, Australia's largest grain exporter, AgriDigital completed a pilot that focused on matching title transfer of a grain asset to payment, as well as supply chain provenance and traceability of organic oats.

This pilot and the design and architecture decisions are discussed in greater detail below in Section 12.3.

Proof of Concept: Rabobank
December 2017, Sydney, NSW

AgriDigital teamed up with the world's leading agricultural bank, Rabobank, to conduct a proof of concept that successfully demonstrated a purchase and sale of commodities on a blockchain in a lab environment.

The objective was to test whether the AgriDigital platform, supported by a blockchain, could facilitate purchase and sale transactions. The proof of concept simulated the execution of a commodity purchase and sale transaction in the form of a Rabobank structured inventory product (SIP), with automated settlement of the purchase and forward sales contract, involving three parties: a farmer, a grain trader, and Rabobank. Under the traditional SIP arrangement, a grain trader enters into an agreement with Rabobank to act as an agent in purchasing grain from farmers. Under the agreement, the trader can purchase the grain from Rabobank within a specified period, at which point the legal title passes to the trader (Fig. 12.4).



PROCESS

- **A** Rabobank issues AUD token to the Facility under Master Agreement.
- **B** Grain delivered and registered at warehouse.
- **C** The Digital Title to the grain is issued to the Grower by Trader as Agent acting as Site Operator.
- **D1** Grower allocates the grain to Purchase Contract with Trader as Agent for Rabobank.
- **D2** Digital Transfer Agreement Executes payment from Rabobank to the Grower in AUD token, ownership of digital title token transfers from the Grower to Rabobank.
- **E1** Trader requests to purchase the grain which is the subject of a forward Sale Contract.
- **E2** Simultaneously Swap Agreement Executes payment from Trader to the Rabobank Facility and beneficiaries (Buyer Digital Wallet and Rabobank) in AUD token. Ownership transfers from Rabobank to Trader.

Fig. 12.4 Overview of the proof of concept with Rabobank. © 2018 AgriDigital, reprinted with permission

On a private Quorum blockchain, the smart contract layer auto-executed the transfer of ownership of the digital title in the commodity from the farmer to Rabobank in exchange for payment made via the buyer's SIP facility. At a later time when the trader was ready to sell the grain to a third-party, smart contracts were used to simultaneously execute the title transfer and settle a number of payments. These payments included repaying the Rabobank facility, passing on interest to the bank and the trader receiving the remaining proceeds of the sale. All payments were made in real time using a Rabobank-backed digital dollar pegged one to one with the Australian dollar.

Using smart contracts to execute the complexities of an inventory finance product goes a long way to automating time-consuming business processes. In turn this

reduces the cost and error rate when making loans under structured inventory products. Traditionally these types of financing arrangements incur substantial back-office costs through numerous title transfer processes and asset valuations including weekly mark-to-market valuation based on third-party reference prices. Financing digital assets backed by live data inputs reduces the risk and cost in executing these financing arrangements. Additionally, using a distributed database to store data on the quality and quantity of grain enables financiers, growers, and traders to access a single source of truth about the commodity, valued in real time.

A highly novel outcome of this proof of concept was the incorporation of a bank-backed digital currency, meaning real-time payment to the farmer was possible in a currency that the farmer recognized and could easily transfer from digital to traditional Australian dollars. Providing stable, digital currencies that are widely accepted by traditional businesses remains a challenge across the blockchain ecosystem globally.

12.3 Designing for a Business Use Case

In this section, we describe the second proof of concept in some technical detail, including requirements, architecture design decisions, and outcomes.

12.3.1 Overview

In July 2017, AgriDigital and CBH Group, Australia's largest exporter of grain, conducted a pilot to test the application of blockchain in the Australian grains industry at CBH's wholly owned subsidiary, an oat processor in South Australia. The pilot comprised two distinct scenarios built on the AgriDigital commodity management platform, blockchain infrastructure and smart contract library:

1. Generating digital title to a physical commodity and executing payment on a blockchain, including secure 7-day payment terms
2. Verifying the organic status of a batch of oats along the supply chain: from a delivery leaving the farm gate, then commingling with other farmers' produce in processing and milling, through to the point of sale to a retail customer

The pilot aimed to cater for a real business use case and therefore needed to integrate seamlessly with existing technology solutions and meet realistic business requirements. This included more accurately reflecting the industry demands, by allowing 7-day payment terms while providing security over the asset for the farmer during this period. It was also important to ensure a level of privacy could be maintained between participants, as the exact processes and procedures on site should not be disclosed to all network participants.

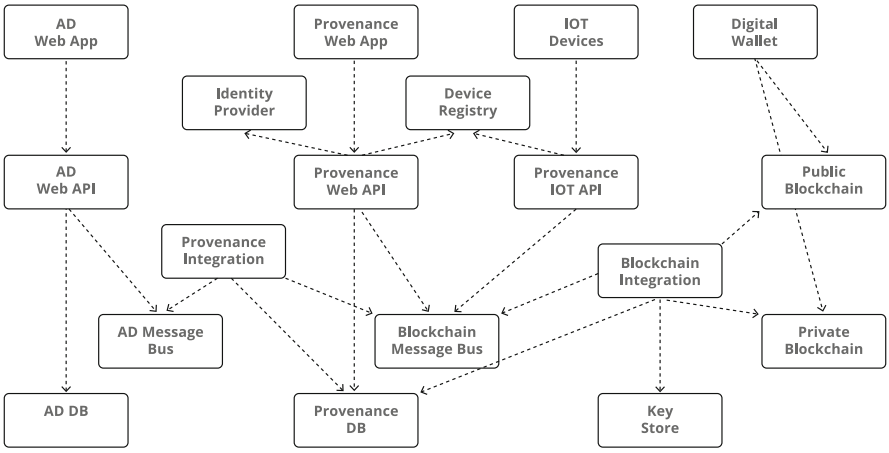


Fig. 12.5 Pilot components (AD, AgriDigital; DB, database). © 2018 AgriDigital, reprinted with permission

12.3.2 Pilot Scenarios

We explain the solution here, before describing the challenges and design decisions in the subsequent section.

Components Based on the protocol design decisions discussed below, the infrastructure shown in Fig. 12.5 was designed for the purpose of conducting the two pilot scenarios. The pilot used the AgriDigital platform and a basic web application as the user interfaces to capture the data. The blockchain protocol ran using a private Quorum network with three physical nodes.

Scenario 1 Transaction and Payment Security Using the AgriDigital platform, digital title to a delivery of oats was generated on a private Quorum network and held in the farmer's digital wallet. Seven days later, settlement occurred in an atomic transaction: payment was made from the buyer to the farmer and simultaneously title transferred from the farmer to the buyer. For the period up until payment, the farmer had clear ownership of the digital title token that represented the physical grain delivery and therefore security over their asset (Fig. 12.6).

The farmer's delivery was received at the buyer's site using the AgriDigital platform, where information around the quantity and quality of the commodity was captured at the point of receival. This information was then pushed through various integrations in order to generate a digital title token on the blockchain. The token was then held and flagged for payment in 7 business days. The payment on the blockchain layer was made using a second token, minted by AgriDigital and known as 'AgriCoin', which was pegged 1:1 with the Australian dollar. Smart contracts, agreements codified for execution on a digital distributed platform, were

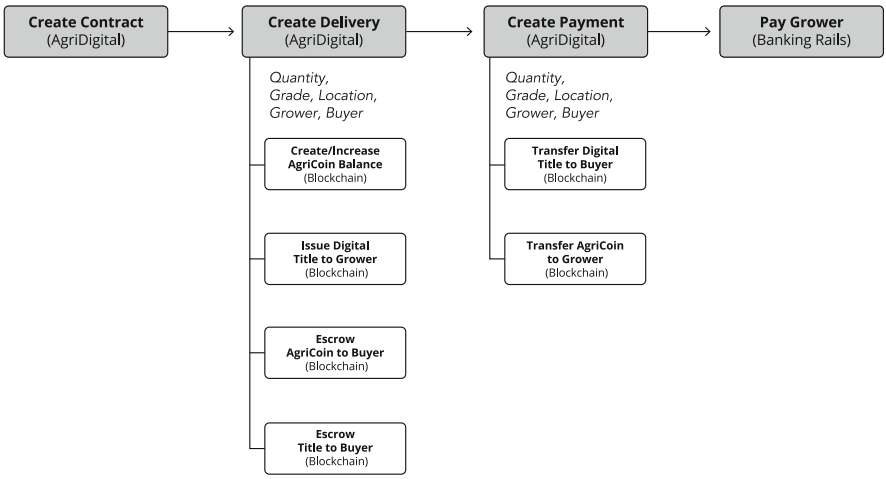


Fig. 12.6 Process flow for grain delivery, sale, and settlement. © 2018 AgriDigital, reprinted with permission

used to auto-execute payment on the blockchain layer. In parallel, the payment was processed using traditional banking methods, using a platform named Sybiz, to ensure the farmer received payment in Australian dollars.

A transaction normally taking days to execute in current grain supply chains happened in less than 1 s under the pilot conditions. This pilot therefore solved the challenge of matching delivery to payment and proved the ability to eliminate counterparty risk by running commodity transactions on a blockchain. This allows the supply chain to operate in confidence: farmers are assured they continue to own their commodity up to the moment they are paid. This goes a long way toward removing counterparty risk along supply chains.

Scenario 2: Provenance and Traceability In a second scenario, AgriDigital and CBH used a private Quorum blockchain to trace the movement of a batch of organic oats from the farm gate, through milling and production, to a retail consumer. Data on the provenance including all intermediate steps was stored and analysed on the private Quorum network.

A range of physical inventory data points were captured on a web application and bundled into assertions, each representing an event or claim determined to be critical to the organic status of the oats. Each assertion was then hashed and recorded on the blockchain layer. At the point of sale to a consumer, the assertions pertaining to that particular batch run were analysed to produce a report that either confirmed or denied the organic status of the oats.

AgriDigital developed an analytics model to determine whether the oats were organic at the farm gate and checked a predefined business process through hulling, milling, and packaging. The analytics model then produced a true/false statement as to whether the organic status had been retained while the batch of oats passed

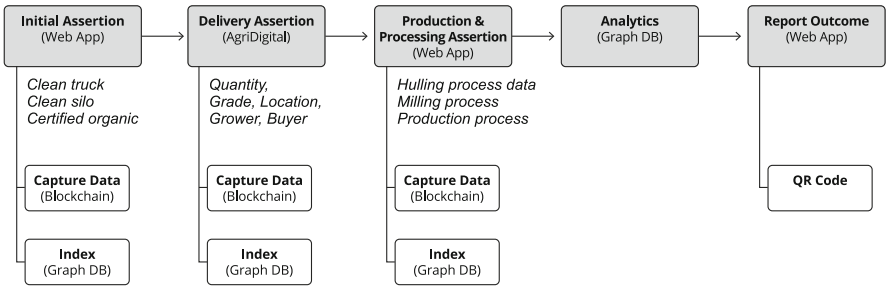


Fig. 12.7 Process flow for analysing the organic status of oats along the supply and processing chain. © 2018 AgriDigital, reprinted with permission

through the various stages of the supply chain to the point of packaging and readiness for delivery to the retailer (Fig. 12.7).

12.3.3 Design Decisions

1. Choice of Protocol AgriDigital had experience working with the Ethereum platform from the first proof of concept. However in designing for a commercially viable use case, there was a strong need to address business concerns around transparency and scalability. The critical technical requirements in designing a solution to meet the business use case were throughput and privacy, and Ethereum was not the best fit. As for privacy, production information around the commodity can be incredibly sensitive and reveal the comparative advantage of individual businesses. However, at the same time, if information or claims need to be proven, the business needs to be able to share this data or otherwise reveal a claim or certificate in a way that the party receiving the information can have complete confidence in its validity. Regarding throughput, at any point in time an enormous volume of transactions are being executed across agriculture supply chains. However due to the nature of the physical transaction, there is a higher degree of tolerance for delay in the transaction time. For this reason, the transaction speed (latency) itself was not as critical as the number of transactions processed per unit of time (throughput). For these reasons, AgriDigital investigated using Quorum as a technology platform for this trial.

The Australian grains industry produces a total of 40–45 million tonnes of grain (cereal crops, primarily including barley, grain sorghum, maize, oats, triticale, and wheat) each year, the majority of which is transacted over intense harvest periods. The Quorum network uses the Raft consensus mechanism with 50 ms block times, meaning transactions can be processed in real time. This allowed AgriDigital to produce sub-second transaction times for the exchange of digital currency and digital title. At a rate of four transactions per second, this settlement method is scalable to satisfy the throughput and latency requirements of the Australian grains industry.

In comparison to solutions available in July 2016, Quorum seemed to offer the most promising solution to business challenges in dealing with the radical transparency of other blockchain protocols. During the protocol investigation stage, tests were conducted on the Quorum chain to determine the utility of the privacy settings in the specified use case. While allowing for some sophistication, ultimately the privacy settings available at the time were inadequate to properly facilitate the business case as required. Using the 'privateFor' parameter, Quorum allowed for transaction details to be made visible only to the parties specified in the contract. However this feature was only available at the level of nodes, rather than between individual addresses. Requiring each participant within the agri-supply chain network to run their own node is not a feasible model in this industry. While Quorum allowed for the transfer of private information in the pilot scenario, it was recognized this was not a scalable or commercial solution for the problem at hand at the time.

2. Network Incentives The scenario was conducted in a closed and controlled network, so there was not a real concern around incentivizing good behaviour by the pilot participants. However, the design of the solution had to cater for the possible scenario where participants did not trust one another and might try to gain an unfair advantage.

The impetus behind including blockchain as part of our technology stack was specifically to allow for unknown counterparties to trust the business transaction. It was therefore critical to design a protocol that incentivized good behaviour and prohibited or revealed bad behaviour wherever possible.

Supply chains are inherently complex, and designs for digital assets must meet the realities of the industry. AgriDigital's understanding of these complexities, particularly in the grains industry with both on-farm and off-farm storage of assets and the various points of sale, directed the design of the systems architecture. Given that the grains industry is largely a handshake industry revolving around physical assets, it was critical to design a network that was capable of giving confidence to the digital asset. Part of this are the incentive structures and network rewards, to ensure good behaviour in the digital ecosystem by the range of participants along the supply chain.

For example, the generation of the digital asset and the issuance of that asset on-chain needed to incorporate means of protection against the potential for fraudulent or deceptive behaviour. It is necessary to separate the issuance of the digital asset from the owner of that commodity, to give confidence that the digital asset was reflective of the commodity in the physical sense. In practice, this means that while the farmer is the owner of the physical asset, it is the warehouse or site operator who takes the quality and quantity measurements and issues the digital asset to the digital wallet of the farmer. By permissioning the different roles of custodian or owner into the network, we are able to prevent bad behaviour. Otherwise an actor could, e.g. issue a digital asset to themselves multiple times without others having any way of verifying whether this reflects physical commodities.

3. Secure Payment Terms AgriDigital's previous work looked at executing smart contracts that facilitate the payment for a commodity at the exact time that delivery was made by the farmer. However, to cater for real business needs, a commercial payment model needs to allow for alternative payment terms. Even where payment for a commodity is not made at the precise time of delivery, the critical requirement is that the farmer continues to own the asset in their digital wallet until the payment is executed.

There were a number of ways this could be managed. AgriDigital decided to employ a fairly straightforward model where both the digital asset and the payment amount were locked or escrowed until the moment the smart contract executed the transactions several days later.

The design of the escrow smart contract was as follows:

1. The buyer adds a pending payee.
2. A smart contract function checks the digital wallet of the buyer to ensure they have the available balance. If the balance is insufficient, an exception is thrown. Returning 'false' is not useful if we are calling the function directly from the blockchain API, as this will only return the transaction hash. Alternatively transaction results could be checked with the hash, but this would be less convenient and require more API calls.
3. If sufficient funds are available, reduce the balance by the amount required, and add it to the escrow balance of the smart contract.
4. Add a new mapping from the payee to the amount requested.
5. Emit an escrow event.

If there is an issue with payment, or the sale is otherwise cancelled during the payment period, the asset or payment amount can be unlocked and returned to the available balance in the digital wallet.

In the escrow function, digital assets were locked using the same functionality. Double sale of assets remains a significant challenge in agriculture supply chains. Revealing a clear and verifiable chain of custody and ownership assists in preventing the problem of double sale, as each digital asset can only be sold once by each party in the supply chain.

4. On-Chain and Off-Chain Data Storage Across the various stages of growing, transporting, and producing organic oats, there are thousands of data points available to be captured. Determining the data points which are valuable to the supply chain as a whole, to individual consumers and those relevant to proving the organic status of the oats, requires close matching of the digital assertions to business processes. In the provenance model, a range of physical inventory data points were captured on a web application and bundled into assertions, each representing an event or claim determined to be critical to the organic status of the oats. The decision was made to store the majority of data points off-chain, while recording the assertion on-chain.

The analytics model was designed to provide a true/false statement as to whether the oats were organic at the point of sale to a retail consumer. Fundamental to making this claim is a model that is flexible enough to cater to a range of use

cases, while ensuring that defined tolerances are not exceeded. Rather than being stored in a contract state, each assertion was recorded on-chain as log events. The decision to store as log events significantly reduces the cost of storing data on-chain. In making this decision, it is important to note that querying log events is much less straightforward than querying contract state.

From the data stored on the blockchain, a graph database (Neo4J) can then be created containing each of the assertions. The choice to use a graph database was made because it provides a flexible model for querying by matching patterns in the graph. The assertions recorded are, by nature, highly variable—in terms of both their associated properties and the connections they draw between entities and artefacts in the supply chain. The flexibility of the graph model was therefore deemed essential in implementing a system that could adapt to the variety of possible scenarios arising in the business case. The final analytics model ran in a web application with access to the graph database.

5. Integrations and the Application Layer Blockchain technology alone does not provide certainty that the digital record reflects the physical commodity. While the digital asset can be highly trustworthy, it is the confidence in the broader data integrity that gives it value. To this end, a commercial solution requires a robust digital infrastructure that connects the physical commodity to the digital representation at every stage along the supply chain. AgriDigital sought to use integrations with platforms and Internet of Things devices, sensors, and machinery such as weighbridges wherever possible.

It was considered whether users could interact directly with the blockchain or whether the interaction had to come through an application layer. Manual, human data input continues to act as a threat to data integrity with poor, incorrect, or incomplete data being hashed to the blockchain. For this reason, the choice was made to interact only through application layers, making the AgriDigital platform and web application both critical components in ensuring that data of a high quality and consistency was posted to the blockchain.

Given the pilot used a trusted network of selected participants, it ran using a private three-node Quorum chain and did not operate distributed components or a clustered graph database. In a completely decentralized system, the Quorum chain would have multiple operators interacting with various platforms and APIs, with access to the same raw data allowing them to make independent verifications that the assertions were correct.

Furthermore, where the blockchain or the application, such as the AgriDigital platform, can integrate with machines and digital systems, such as a weighbridge integration and quality testing instrumentation or processing equipment, this is clearly preferred. Removing human data input and increasing the number of such integrations allows for much more reliable data entry and increases the integrity of the blockchain-backed data overall. Part of facilitating these integrations is increasing digitization across businesses generally in order to build out the entire digital infrastructure.

As a blockchain is an immutable store of data, it is critical to ensure both the data and the actor making an assertion are correctly identified at the time of record

on the blockchain. Digital identity was not a primary requirement for this particular business use case, and therefore an identity component was considered, though not integral, in the design for this pilot. A more robust digital identity solution should be considered in subsequent work.

6. Digital Currency Operating digital currencies across a network remains a challenge. New kinds of 'programmable money' aim to represent fiat currencies and leverage the benefits of blockchain technology in being programmable. Approaches such as 'stablecoins' attempt to provide stable exchange rates with fiat currencies but were only in their infancy and were not considered.

Three methods of payment were considered for this pilot:

1. Payment in cryptocurrency
2. Payment in a network token
3. Payment off-chain using traditional banking methods

For the purpose of the business use case, cryptocurrencies such as Bitcoin were considered to be too volatile and difficult for businesses to hold on a balance sheet. Without the existence of a centrally issued Australian digital dollar, it was not feasible to consider making payment in a cryptocurrency.

Payment on a blockchain can be significantly more efficient as it occurs in real time and has the benefit of being programmable. This means complex sequences of events and dependencies can be written into the blockchain itself to allow certain events, such as matching title transfer to payment and automating financing arrangements, within a single atomic transaction.

However, in designing for a commercial use case, it was critical that the farmer received payment in a currency of value to them. Therefore a decision was made to execute the payment on-chain in a network token, AgriCoin, and make a parallel payment using traditional banking rails. This allowed the smart contracts to leverage the programmable nature of digital currency through the AgriCoin and allowed the farmer to receive payment in a currency of value to their business, being Australian dollars. Payment on a blockchain using digital currency payment was proven to be significantly more efficient, as it occurred in real time and had the benefit of matching title transfer to payment. However, the trade-off in this model was that the full benefit of this efficiency was not realized, and risk reintroduced, by moving off-chain to make the final payment in Australian dollars.

12.4 Summary

With global caloric demand said to increase by 70% by 2050,[7] the agriculture industry is only growing. AgriDigital is building a piece of the digital infrastructure to support the development of digital trust across agriculture supply chains. The

[7]FAO Synthesis Report, 'How to Feed the World in 2050', 2009.

AgriDigital blockchain protocol aims to be a low-cost product accessible to all participants across the supply chain.

The AgriDigital pilot program has continued to test the logic and technical solution. Each iteration has included advances in the technology stack and more complicated use of smart contracts to facilitate novel trade and finance arrangements. Key challenges for the AgriDigital blockchain product going forward include solving for digital identity, data integrity, and business privacy requirements.

Some commentators claim that in a few years' time blockchain will no longer be a buzzword—it will be as ubiquitous as the internet. Others believe that blockchain is all hype; that it is an untested technology with huge risks and little upside. Farmers have always been eager adopters of technologies that make sense and deliver real value. AgriDigital is already starting to realize that value for farmers and agri-supply chain participants, with a clear vision to incorporate blockchain as part of the solution to solve the big challenges in agriculture supply chains.

Chapter 13
Case Study: SecureVote

Taking a Dapp from MVP to Production

with Max Kaye and Nathan Spataro

'Democratise the world.'
— SecureVote's Massive Transformative Purpose (MTP)

13.1 Introduction and Background

Voting seems simple enough. With paper, voters just fill out a ballot sheet and put it in a box. To count the votes, the box is emptied and the ballots are counted in public. However, there are many underlying complications. How do we know extra votes have not been added? How do we know each voter has voted at most once, or exactly once? If a voter claims their vote *was not* in the final tally, how could we check? How do we know the count is accurate, especially if it can vary every time votes are counted?

Solving all of these problems can be hard, even with paper systems. With electronic voting systems, some things become easier. For example, there may be a public electorate-wide list of voters, and we could ensure each vote has some kind of verifiable cryptographic authentication. This can help us check to who did not vote. And, of course, tallying votes is fast and reliable in an electronic system. However, electronic systems present us with other problems. How can we anonymize votes in an electronic system? How do voters know whether their vote was included without revealing who they voted for? How can we decide which votes are valid without a privileged role?

The potential utility of blockchain technology for voting was identified early,[1] and blockchain can help to solve some of these problems. However, using a blockchain alone is not enough. The exact blockchain chosen, the consensus mechanisms used, and the architecture of the voting platform are all important design decisions

[1] At least by December 2010: https://bitcointalk.org/index.php?topic=2299.msg31851#msg31851.

X. Xu et al., *Architecture for Blockchain Applications*,
https://doi.org/10.1007/978-3-030-03035-3_13

that impact system capabilities. While this chapter does not describe solutions to all of the hard problems with online voting, it does describe architectural concerns for robust, upgradable, multi-user smart contracts used by SecureVote to address these problems.

This case study concerns the development of *Tokenvote*, a general purpose, multichain governance system for blockchain-based tokens developed by *SecureVote*. Tokenvote uses the Solidity language and is deployed to Ethereum.[2] Tokenvote supports arbitrarily complex append-only voting systems and has been designed to be modular, upgradable, and configurable. SecureVote was founded in 2016 to provide affordable, turn-key voting systems of all types and at all scales.

In this chapter we will cover many of the challenges encountered and trade-off decisions made while taking this smart contract-based voting solution from a minimum viable product (MVP) to production. This journey spanned 4 months, numerous redesigns, and all of the ecosystem issues mentioned above. We will not describe the voting functionality in detail—the principles are outlined in the sidebar below—but rather focus on how the architecture and overall design changed over development. The discussion is in terms of the Solidity language on Ethereum, but many of the architectural issues apply across smart contract languages and platforms.

Principles of Anonymous Voting Using Blockchain

Through a combination of public-private key cryptography and peer-to-peer shuffling, SecureVote achieves that voters can vote anonymously and later verify and confirm that their vote has been recorded correctly. Voters cannot prove which vote was theirs, and no one else can find out how they voted.

To achieve these goals, a ballot is prepared with an electoral roll, containing all addresses that are allowed to vote. For this ballot, voters first create and anonymize an *ephemeral* voting key pair, which is discarded after the ballot completes. The voters use this key pair to anonymously sign their actual vote.

Two rounds of shuffling are necessary: the first one to create the ephemeral anonymized key pairs and the second one for the actual voting. In each round, the shuffling is done *off-chain*, in a peer-to-peer fashion but relying on *on-chain* information like the electoral roll; and the results of the round are published *on-chain*. After each round completes, each voter confirms that the result is well-formed and that their vote/ephemeral public key was recorded correctly, by signing the result.

(continued)

[2]Links to the Tokenvote source code and Ethereum documentation appear in Section 13.6 at the conclusion of this chapter.

Say there are 50 voters. The ballot ensures that each voter can vote at most once and only voters on the anonymized electoral roll can vote. If all 50 voters confirm that their individual vote was recorded correctly, then we know that *all* votes were recorded correctly. In other words, the number of signatures on the ballot must match the number of voters exactly.

More details can be found at https://gitlab.com/exo-one/svst-docker/blob/master/svst-docs/secure.vote.white.napkin.md.

13.2 The MVP Prototype

In late 2017, SecureVote implemented a small MVP to facilitate early governance for the US-based *Swarm Fund*, a blockchain-based organization facilitating the creation of securitized tokens. Although Swarm Fund's security tokens live on a Stellar-based blockchain, their organization-wide token (SWM) is an ERC20 token on Ethereum.

Swarm (unlike many ERC20-based organizations) were proactive about governance from the start. In their whitepaper they described the first version of their *Liquid Democracy Voting Module* (LDVM), a system designed to support the governance of both the foundation and the investment opportunities offered via their platform.[3] There are two important aspects of their design that are common in systems of distributed governance: delegation and stake-weighted votes.

- *Stake-weighted votes*: In many ballots, not every vote is weighted equally, or some parties may have an unequal number of votes. The most common example of stake-weighted voting is by shareholders at a company's annual general meeting (AGM). Each shareholder votes with a weighting proportional to the number of shares they own: 1 share, 1 vote; 2000 shares, 2000 votes. For similar reasons, most token-based communities choose to use stake-weighted votes.
- *Delegation*: Voters can choose another party to act on their behalf. On a blockchain, this could be another account they own, for example, allowing voters to delegate voting power from tokens they own in a cold wallet to a 'voting-only' account in a hot wallet. Or, the delegate could be someone else's account, for example, a prominent community member. Delegation is a common feature of modern digital governance systems. It is similar to the idea of a representative in government but can be done on a per-voter basis. In some systems multiple delegations can be chained together. The original voter can always stop the

[3] Swarm's LDVM design uses fairly standard patterns, and the requirements are currently met by a subset of Tokenvote's capabilities.

delegation and vote directly; only if a voter *does not* vote does the delegate inherit the voter's weighting.

SecureVote was responsible for the initial implementation of Swarm's governance framework. This initial deployment had only a few requirements:

R1.1 Facilitate an open ballot for all SWM token holders and all delegates.
R1.2 Support optional delegation to arbitrary Ethereum addresses.
R1.3 Stake-weight votes according to voters' SWM balances and delegations.
R1.4 Support the deterministic audit of the ballot by arbitrary actors.
R1.5 Support the encryption of votes such that the result is unavailable until the respective secret key has been published.

These requirements seem simple but are practically impossible to meet using *only* smart contract platforms like Ethereum and *on-chain* computation, where all storage, auditing, and delegation resolution occurs within the blockchain's virtual machine. There are two primary reasons for this:

- Historical access: a naive voting system might check a voter's balance at the time the vote is cast. However, this approach introduces multiple race conditions and makes handling delegation difficult. The correct approach is to use snapshots at the start and end of the voting period to retrieve balances and delegations, respectively. Ethereum does not support this kind of arbitrary historical access.
- Transaction cost: with fee-per-operation blockchain platforms, like Ethereum, it is prohibitively expensive to repeatedly load items from storage (like balances, delegations, and votes) and run tightly looped algorithms such as recursive delegation resolution or vote counting. As an example: a well-tuned smart contract could process a maximum of around 400 votes per Ethereum block, or 1600 votes per minute, based on a gas limit of 8 million gas. Processing greater volumes requires splitting the operation across multiple transactions, a tactic which adds overhead and code complexity. Some of the more interesting features, like on-chain decryption, are simply untenable under fee-per-operation models.[4]

SecureVote has previously argued that secure voting (be that on paper or online) is impossible at scale without the use of a well-constructed blockchain. This is due to three goals:

- Immutability: the voting record must be append-only and cannot be changed (even if individual votes can be replaced).
- Censorship resistance: no actor should be capable of preventing a voter from submitting their vote, except through violence. This requirement precludes purely proof-of-stake chains in most cases.

[4]Note: computations like on-chain decryption can be more practical with protocol-layer optimizations such as the `ecrecover` function supported by Ethereum.

- Consensus: voters and auditors must agree on which votes are to be counted *both during and after* the voting period, and these rules must be non-authoritarian (due to the need for censorship resistance), objective, and non-discriminatory.

All centralized systems (including recent end-to-end verifiable designs such as *Prêt à Voter*) fail at least one of these requirements (usually censorship resistance) and are thus not fully secure.

Although Requirement R1.4 (deterministic audit) requires a blockchain to *store* vote data, it does not require that the audit itself is performed on-chain. The lack of historical access available to smart contracts[5] meant SecureVote needed to audit the ballot off-chain. Given this, we opted to move as much processing and functionality off-chain as possible without compromising the platform's integrity. Decryption of votes was done every time an audit was run.

The initial MVP was incredibly simple, with only three components:

- A small smart contract of around 100 lines of Solidity code, to securely deliver ballot details and store votes
- A rudimentary auditor to authenticate voters, decrypt votes, allocate the appropriate weighting, and resolve delegations
- A user interface

At this stage, the MVP was unable to handle multiple ballots or communities, and an individual smart contract had to be deployed for each ballot, costing around 800,000 gas at a minimum. Although this rudimentary system was quite capable of handling Swarm's needs for the next few months, it was unsuitable for general use and required costly manual attention for every deployment.

13.3 Building Tokenvote

Although the MVP was functional and satisfied basic requirements for one-off ballots, it was not a fully fledged product. Prior to February 2018, SecureVote intended to launch their platform via a custom, separate blockchain they had been developing since June 2017. Although development had been progressing steadily (two prototypes existed at this stage), it was not progressing quickly. In order to launch a viable platform in the shortest period of time, they made the decision to pause development of their custom chain, and pivoted to building out the MVP into a general software-as-a-service (SaaS) platform: Tokenvote. This was to reduce development time and support most of the features of their custom chain, albeit with reduced capacity.

This section covers many of the problems SecureVote encountered while building Tokenvote based on the MVP described above. For each problem we will look at one

[5]Ethereum smart contracts have access to the past 256 states only (corresponding to the past 256 blocks), a period of approximately 1 h.

or more potential solutions and discuss compromises. Some simplified Solidity code is used in the presentation.[6] The simplifications are made to keep the examples as short as possible, so best practices are sometimes ignored.

As a more generic platform, there were additional requirements:

R2.1 Centrally manage and track groups (democracies), including Ether payments and permissions.

R2.2 Allow group administrators to create new ballots, and control permissions around ballots from community members.

R2.3 Extensibility and maintainability: any component can be upgraded, new components can be added, and Tokenvote must support migration to another platform in the future.

R2.4 Browser compatible: the whole stack should be able to be run in a browser, excluding the Ethereum nodes themselves, without compromising the security model.

The goal was for Tokenvote to facilitate everything the Swarm prototype did and more, but to cater for many groups, each with many ballots, without needing any interaction with SecureVote staff.

13.3.1 Tokenvote Architecture Overview

The initial, planned architecture for Tokenvote is shown in Fig. 13.1. After numerous iterations the final architecture is as shown in Fig. 13.3. Each is discussed below.

Planned Architecture

In the initial architecture of Fig. 13.1, administrators interact with an on-chain component that serves as a central hub, which SecureVote call the `Index`. This component is responsible for all administrative functions, including payment of fees for holding a ballot. The `Index` also keeps track of groups of voters, called *Democracies*. If fees are paid, a ballot is set up for a Democracy through a factory contract, the `Ballot Box Factory`. See Section 7.4.4 for a general discussion of the factory contract pattern. This factory contract can create a ballot by deploying a new `Ballot Box` smart contract, through which voters can cast their votes.

For the reasons explained around the MVP above, tallying and weighting of the votes is done offline through an `Auditor` component. This component is available to any voter, so that independent auditing is possible. The `Auditor` also queries the relevant ERC20 contract for token holdings and other details as required, including

[6] Simplifications include omitting keywords like `view` or `pure` on function declarations, and declaring functions `public` instead of `external`.

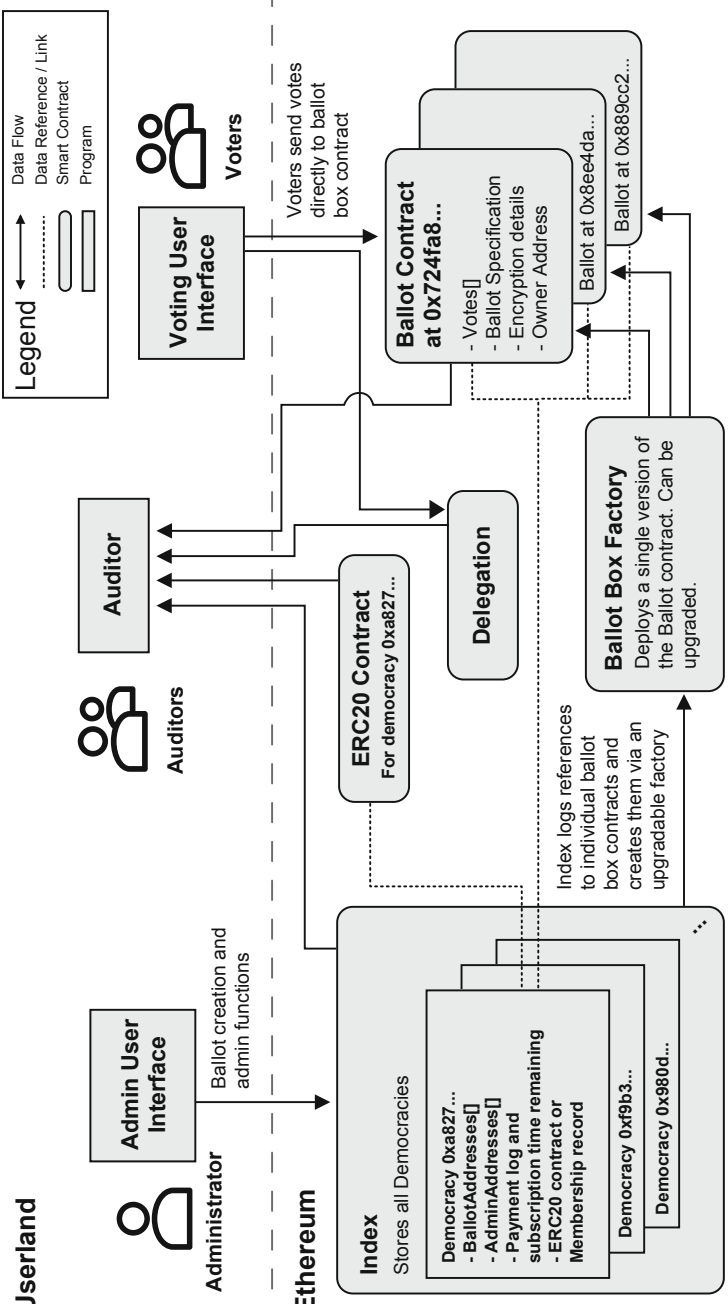


Fig. 13.1 Planned architecture for Tokenvote before development

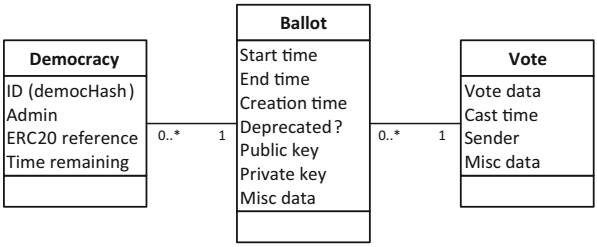


Fig. 13.2 Logical view of the required data structure, as a UML class diagram

delegation and balances on other chains. This allows the auditor to consider tokens across the public Ethereum and Ethereum Classic blockchains, among others.

The logical data structure for this design is depicted in Fig. 13.2. As shown, each democracy can have an arbitrary number of ballots, and each ballot can have many votes. Each vote belongs to one ballot and each ballot to one democracy.

The reference to the relevant ERC20 contract is stored for the democracy. Encrypted ballots can be held by generating a key pair, publishing the public key for all voters to encrypt their votes, and revealing the private (secret) key after the end time of the ballot. This method avoids influence of intermediate results on voters while the ballot is ongoing.

Final Architecture

In the final architecture depicted in Fig. 13.3, the basic components are still present, but there are some significant changes:

- Instead of creating one smart contract per ballot, all ballots that use a particular feature set are stored in the same contract, the `Ballot Box Storage`. This includes ballots from different democracies. It in turn relies on the code outsourced to the `Ballot Box Library` contract, implementing the data contract and library contract patterns from Section 7.4. Collapsing all ballots into few smart contracts is more efficient in terms of gas cost. As discussed in earlier chapters, this results in reduced monetary cost, increased throughput, and reduced danger of network congestion.
- Following the same patterns to achieve upgradability and separation of concerns in the `Index`, data on payments is stored in the `Payments Backend`, and all other data for the `Index` is stored in the `Data Store Backend`. Pricing for community ballots is calculated in the `Community Ballot Payment` contract; adaptive, context-dependent pricing is needed to avoid spamming democracies with too many ballots.
- To allow easy addressing, the Ethereum Name Service, `ENS`, is used. The `ENS Proxy` implements the contract registry pattern (Section 7.4.1). Requesters can look up the reference for the latest version of the `Index` contract.

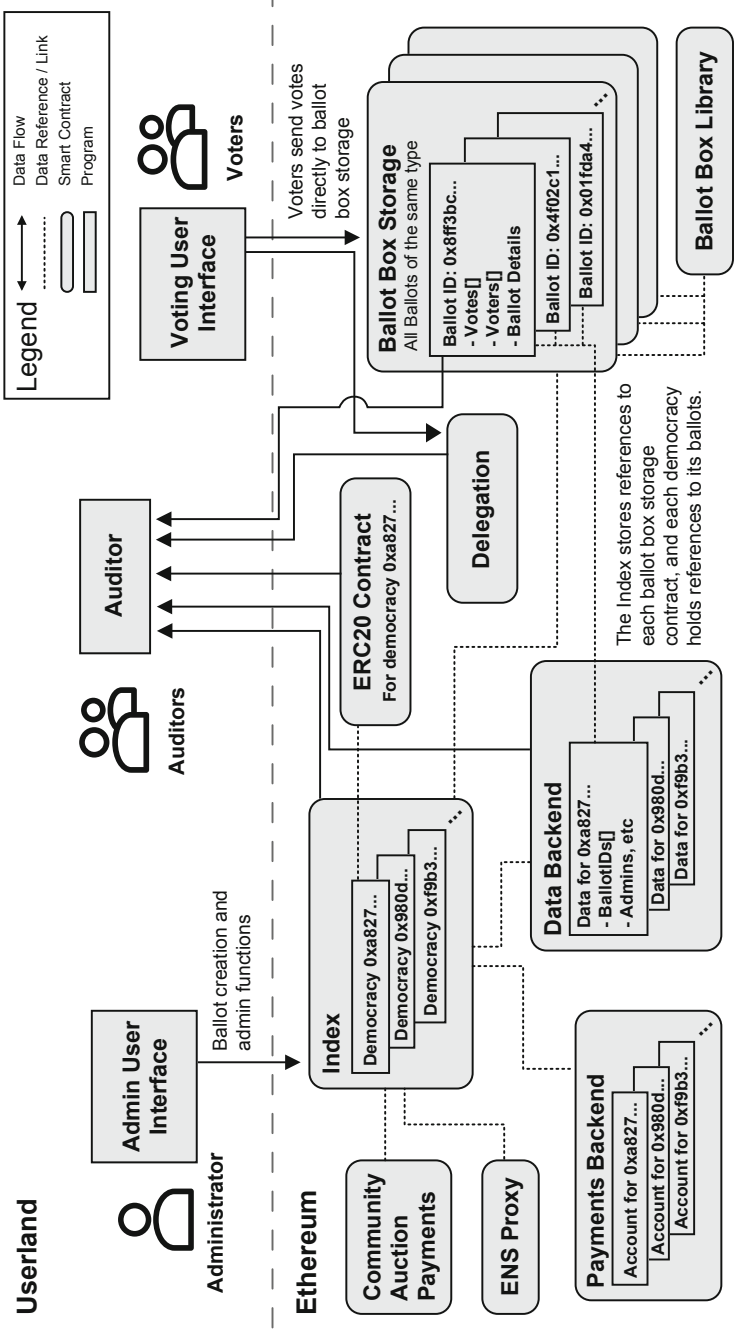


Fig. 13.3 The architecture for Tokenvote at deployment. Many alterations were needed before SecureVote considered the dapp production quality

In Chapter 5, we discussed ways in which blockchain can be used architecturally. In the Tokenvote architecture, we note the following:

- Tokenvote uses the blockchain as a storage element, as a communication mechanism for publishing ballots and votes, and as an asset management and control mechanism for payments and checking stakes.
- The use of blockchain as a computational element in this architecture is limited. Most computation is done off-chain, for the reasons outlined earlier. On-chain computation serves primarily to enforce checks such as authorization, what to store, and hash integrity. Other smart contract codes implement schemas for data, particularly ballot data stored in the ballot box storage. Not all data schemas are implemented in smart contract code, to allow for more flexibility in schemas that are immaterial to the core concerns of the solution.
- Regarding the integration of blockchain into a system as a component, this architecture is rather interesting, in that there are only in-browser components in addition to blockchain. However, running the auditor in-browser relies on SecureVote's full blockchain nodes, which introduces some level of trust on their integrity and truthfulness. Alternatively, anyone could host their own full node to function as an auditor. However, an auditor requires the full history including all states, which prevents syncing the blockchain through fast-sync, and requires over 1TB of SSD space at the time of writing.

SecureVote decided to use Ethereum as a technology platform for two main reasons. First, Ethereum's ecosystem was the most attractive, especially the support for ERC20 tokens. Second, despite its limitations, Ethereum was the best available option in terms of security, the execution environment, and the network. To benefit from lower fees, Tokenvote contracts can also operate on Ethereum Classic: transaction fees in fiat currency were a factor of 10–30 lower during the development of Tokenvote.

Qualities and Trade-offs

The common blockchain trade-off is between *transparency* and *confidentiality*, and this is present in Tokenvote. How voters voted needs to remain confidential, but each voter needs the transparency and certainty that their vote has been counted. Out of the options for how data can be stored (discussed in Section 6.3.3), SecureVote decided to use smart contract variables. This was to (i) avoid the need for offline interpretation as much as possible and (ii) ensure integrity, since, e.g., logs are computed in each full node and are not directly part of consensus.[7]

Cost, as discussed in Chapter 9, played an important role. Gas cost, complexity bounds, and limitations of the platform and their impact led to the revision of certain

[7] https://ethereum.stackexchange.com/a/1309.

design decisions, such as collapsing all ballots into one smart contract and storing most ballot details (e.g. title, description, and options) off-chain.

In terms of *performance*, discussed in Chapter 10, throughput plays the most important role. The latency requirement is that feedback to users confirming the recording of their vote should be given within reasonable time, on the order of 1–2 min. For both throughput performance and cost reasons, SecureVote minimized the complexity of voting transactions.

Dependability and *security* concerns (Chapter 11) are of course very important for a voting platform. In terms of *availability*, the most impactful issue would be transactions that are not included. This concerns primarily new or upgraded contracts and the transactions deploying them, since these transactions can incur high gas costs. This risk has partly been mitigated by collapsing all ballot contracts into a few reused contracts. *Reliability* is prominent when running full nodes with full history over a long time, due to high network load and high requirements on fast and sizeable disk space. *Maintainability* and *upgradability* are addressed using the patterns discussed throughout this chapter. *Safety* in the Lamport-Alpern-Schneider sense (see Section 11.3) is addressed through good coding practices and thorough testing with close to full code coverage, including negative tests that test failure cases. In terms of *integrity*, the solution relies on the strong, inherent integrity features of blockchain, and on implementing tight authorization checks for all functions. To ensure integrity for the stake weighting, stake holdings are taken from before and after each ballot. Also, the `Auditor` components ensure that all votes are counted. Auditing starts only 15 min after end of a ballot, which corresponds to approximately 60 confirmation blocks.

13.4 Details and Code Samples

In the following, we discuss some of the issues and lessons learned in detail and provide code samples where they are helpful.

13.4.1 Indexing and Externally Accessing Data

SecureVote's earliest component for Tokenvote was the multi-democracy framework (the `Index`), which would allow ballots to be created within a *namespace* that only the democracy's administrators had access to. Each time a ballot was to be created, a new `BallotBox` smart contract would be deployed. To begin, the voting smart contract MVP was reused, but a different approach was ultimately required.

```
// contract: Index
mapping (bytes32 => Democracy) public democs;
bytes32[] public democList;

struct Democracy {
  address erc20;
  address admin;
  Ballot[] ballots;
}

// return the number of democracies
function getDemocN() external view returns (uint) {
  return democList.length;
}
```

Listing 13.1 Referencing rich data types via unique IDs. SecureVote still uses this pattern today

Initially, each democracy had a unique identifier, via a hash generated from a number of parameters.[8] Unique identifiers are important because they facilitate cheap lookups via arrays or mappings.

In this case, SecureVote stored democracies in the `Index` as in Listing 13.1. This code sample uses patterns that are important when upgrading smart contracts. Since blockchain data cannot be moved easily during an upgrade, it should either be stored in a separate data contract, following the data contract pattern (Section 7.4.2), or remain in the older version of the smart contract and locked from further mutations. This is a direct consequence of Requirement R2.3 and the nature of smart contracts. For Tokenvote, we adopted the latter solution.

Note that in Listing 13.1, the array `democList` keeps an index of known keys, and the function `getDemocN` returns the number of democracies. These provide means for external users to discover information about the contract. Solidity does not have primitives for this nor otherwise directly supports discovery of this kind of information, so developers have to implement it specifically. The combination of this mapping, array, and length getter means that it is possible for everything but the ballots (stored in `Democracy.ballots`) to be easily read externally.[9] Reading the `Ballots` in each `Democracy` requires another set of getter functions (shown in Listing 13.2).

Accessing all data in a smart contract is an important prerequisite to ensure upgrade paths are available. For this reason, important state variables (for the most

[8]Choosing multiple parameters, particularly parameters outside of the user's control, is important to avoid collisions when using this technique.

[9]Technically it is always possible to read any arbitrary data stored on a blockchain at some level; in the case of Ethereum and Solidity, the curious reader can find out about accessing arbitrary variables in contract storage here: https://medium.com/aigang-network/how-to-read-ethereum-contract-storage-44252c8af925.

```
// contract: Index
struct Ballot {
  bytes32 ballotSpec;   // hash of the ballot specification
  BallotBox ballotBox;  // external smart contract reference
}

function getBallotsN(bytes32 democID) public returns (uint) {
  return democs[democID].ballots.length;
}

function getBallot(bytes32 democID, uint ballotID)
  public
  returns (bytes32, BallotBox)
{
  Ballot memory b = democs[democID].ballots[ballotID];
  return (b.ballotSpec, b.ballotBox);
}
```

Listing 13.2 Accessing nested dynamic elements in arrays and mappings

part) require external getters. This allows other smart contracts to read the complete state and helps maximize upgrade potential.

13.4.2 Splitting Up Contracts

Design patterns can increase code size. In general, adding an external function has a very low runtime overhead. However, the additional space required can easily be hundreds of bytes, depending on the number of arguments and data returned. Although this will usually be inconsequential, it can cause issues due to many blockchain platforms setting limits for the size and deployment cost of smart contracts. In Ethereum (due to EIP-170[10]) smart contracts are limited to `0x6000` bytes (approximately 24 KB). Refer also to the deployment risk of large contracts mentioned in Section 11.6.

In SecureVote's case, `Index` grew rapidly during development and hit Ethereum's deployment limit. Managing the size of smart contracts is necessary for any complex dapp deployed to Ethereum and similar networks. In this section we show three approaches: using auxiliary contracts for non-core but useful operations, using tightly coupled 'backend' contracts to offload storage and data processing, and using libraries to allow common code to be used by multiple contracts and hosted separately on the blockchain.

[10] https://github.com/ethereum/EIPs/blob/master/EIPS/eip-170.md.

Augmenting Smart Contraction Functionality via Auxiliary Contracts

Often, smart contracts serve two purposes: storing data and processing that data. For example, a ballot box smart contract might include logic for storing, tracking, and retrieving individual votes. However, to avoid separate `HTTP` calls to retrieve each vote, it makes sense to try and batch these requests to return all votes at once. A straightforward approach would be to add this function to the ballot box smart contract itself. Rather than just the function for individual votes, `getVote(uint voteID)`, we could add another function `getAllVotes()` to return all votes, and functions such as `getAllVotesFrom(address voter)` to return all votes from a particular voter. However, adding such functions increases contract size and introduces complexity to the integrity-critical primary contract.

As an alternative approach, we can use auxiliary contracts. In this example, the primary contract would retain the individual `getVote(..)` function. However, `getAllVotes()` would not be added to the primary ballot box contract. Rather, we create a second contract with a function `getAllVotes(BallotBox bb)` which calls `bb.getVote(..)` for every vote and returns a corresponding array. A single auxiliary contract can work with every instance of the primary contract (in this case `BallotBox`).

This technique has significant benefits: the auxiliary and primary smart contracts are only loosely coupled, so the auxiliary contract can be more easily upgraded or deprecated; code for complex data processing is moved out of the primary contract, reducing testing and attack surfaces; and there is greater flexibility around the type of data returned.

SecureVote uses auxiliary contracts for several purposes:

- Delegation between voters and self-delegation across network boundaries
- Retrieval and preprocessing of votes
- As a lookup table for human-readable names

Adding a Backend Smart Contract

Sometimes a large, interconnected contract must be broken up. We discuss several methods for this purpose here.

The simplest approach is to separate the data storage and longer-term data processing into different contracts. Exactly what is split between ‘frontend’ and ‘backend’ contracts should be based on an assessment of what functionality is likely to be more stable in the long term and what is likely to be more frequently upgraded or replaced. There is a small performance cost to this approach. As a smart contract is split up, it will need to know about the backend contract (and load its address from storage), and the backend contract will need to grant permissions to the frontend contract and verify these permissions. Each call between contracts will also incur some additional fee for invocation and parameter passing. So, this

approach is valuable for infrequently called functions (such as democracy creation) rather than for frequently called functions (such as voting).

In Tokenvote, the `Index` is split in this manner. For example, when democracies are created (and administrated), most of the operations (like calculating the unique ID, storing data, and setting initial permissions) take place in the backend. Only minimal functions are left in the frontend contract. As mentioned, a consequence of this approach is that two sets of permissions must be verified. The first is for the user when calling the frontend contract, and the second is for the frontend contract when calling the backend.

This approach has some nice properties. More than one `editor` can be added, provided the backend authentication has been architected appropriately. This means an alternate frontend can be introduced, or the original frontend swapped out, augmenting or introducing functionality.

Tokenvote required two dedicated backends and several other supporting smart contracts all using this approach. Examples are Tokenvote's ENS (Ethereum Name Service)[11] integration and an auction system for publishing a particular kind of ballot. The latter contract is currently simply a placeholder for future functionality.

Using Libraries

Another way to split up functionality and code is to use a library. These are deployed like Solidity contracts but cannot be called directly and have no state of their own. Rather, they are 'linked'[12] (similarly to the way libraries are linked in `C`) and allow the library to modify the state of the contract calling it. This is particularly useful for logic repeated in multiple contracts. Examples of these sorts of libraries are in the OpenZeppelin[13] framework, which includes many code examples, base contracts, and useful libraries, such as the ubiquitously used `SafeMath` library that has safety checks on inputs.

SecureVote uses libraries in a few specific cases. First, a library is used to handle code that needs to be identical across contracts, such as extracting data from packed variables. Second, a library is used to version and manage the handling of votes and ballots, leaving the *container* contract (which holds many votes and ballots) with a cleaner and simpler codebase. Third, libraries are used like macros across multiple contracts to wrap common multistep operations. Only the first two uses reduce the calling contract's size.

[11] A simple name system has been implemented over Ethereum allowing names like `data61.eth` to be registered and resolved to an address (in the same way domain names resolve to an IP address). In this case the Tokenvote `index` is resolved via `index.tokenvote.eth`. More information can be found at https://docs.ens.domains/en/latest/.

[12] Libraries are used via the `delegatecall` operation, which means 'run this code as if it were inline here and give it direct access to my storage'.

[13] https://openzeppelin.org/.

Libraries are not as simple to upgrade as contracts, and while this is technically possible,[14] it requires preparation and a deep understanding of the underlying blockchain. It can be easier and safer to upgrade an individual contract linked to a new library, rather than the library itself.

13.4.3 Upgrades and Trade-offs

Recently, there has been increased interest in upgradable smart contracts. There are many reasons to upgrade, including to add functionality, to mitigate potential attacks, or to fix bugs. In this section we will describe two techniques used by SecureVote which, when used together, allow for atomic upgrades without downtime for other smart contracts. We also describe how SecureVote plans to improve their upgrade procedure to expand atomic interactions to all users and eliminate any sorts of race conditions entirely.[15]

We start by looking at a simple case of replacing an existing contract. Then, we examine the upgrade of a prototype delegation contract and how SecureVote currently manages upgrading the `Index`. Finally, we describe an oversight and how SecureVote plans to address this.

Replacing Smart Contracts

When the interface to a smart contract is well known, it is trivial to replace it at a future date. A simple way to upgrade a contract factory is shown in Listing 13.3. There are three smart contracts here: an instance of `Frontend` and two instances of contracts which implement the `Backend` interface. The `Index` instance stores a reference to a `Backend` implementation, and upgrading is as simple as replacing this reference. This method is very general and forms the foundation for other methods discussed below.

At compile time, Solidity knows about the interface of remote smart contracts, to generate logic for communicating with them and to check type safety. At runtime, these checks are not performed. In Listing 13.3, the `doUpgrade` function accepts an address `newBackend`, and after the contract is deployed, the owner is free to call `doUpgrade` with any address, including that of a smart contract which does not adhere to the `AuxContract` interface. We can use this lack of runtime checks to implement upgrade schemes.

[14] One way to implement upgradable libraries: https://blog.zeppelin.solutions/proxy-libraries-in-solidity-79fbe4b970fd.

[15] Although well-designed smart contracts can interact with Tokenvote atomically, a race condition exists where an upgrade could take place between the creation of a transaction to the index and the inclusion of said transaction. In this case the transaction would only affect the old index, and would revert in most cases as the index would have lost permissions to modify data on the backend.

```
interface Backend {
  function replaceWith(AuxContract newExternal) public view;
}

contract Frontend {
  Backend _backend;
  address owner;

  constructor(Backend initBackend) public {
    _backend = initBackend;
    owner = msg.sender;
  }

  function doUpgrade(Backend newBackend) public {
    require(msg.sender == owner);
    // let the current _backend know we are upgrading, if
        needed
    _backend.replaceWith(newBackend);
    _backend = newBackend;
  }
}
```

Listing 13.3 A simple, general way to replace smart contracts

SecureVote's First Upgrade

Delegation in voting systems is usually straight forward. A voter can choose someone else (the delegate) to act on their behalf, and if the voter abstains the delegate's vote is used instead. So each delegate votes with the combined power of all their delegators (the voters doing the delegation). Many systems of delegation also include delegation by categories or similar ways for voters to choose one of multiple delegates depending on context.

Less than a month after deploying their first delegation smart contract, SecureVote found they had made an oversight. Although users could make and check delegations, there was no way to iterate through them, and there was no way to find delegators given some delegate. Although, functionally, delegation only works in one direction (where a voter chooses a delegate), *resolving* delegations is more complex. There were two complications:

- In standard ERC20 implementations, there is no complete list of account holders. This means it is impractical to iterate through all potential voters to check their delegations.
- If the voter abstains, only the delegate's vote and address are known. Without delegation backlinks it is not possible to efficiently find the voters who have selected a particular delegate.

```
contract Version1 {
  mapping (uint => bytes32) data;
  function getData(uint i) external returns (bytes32) {
    data[i];
  }
}

contract Version2 {
  mapping (uint => bytes32) data;
  Version1 prevContract;

  constructor(Version1 _prev) public {
    prevContract = _prev;
  }

  function getData(uint i) external returns (bytes32) {
    bytes32 r = data[i]
    if (r == bytes32(0))
      r = prevContract.getData(i)
    return r
  }
}
```

Listing 13.4 The general pattern of a layered upgrade

Since looping through (and caching) all historic delegations was not particularly elegant, SecureVote opted to upgrade their delegation functionality by implementing a second delegation contract which operated 'over the top' of the first. Only when this new contract could not find a delegation would it check the original contract. Since SecureVote references most contracts via ENS names, the upgrade was a simple matter of deploying the new contract and updating the ENS resolution. This general pattern is shown in Listing 13.4.

This simple pattern is very useful in the right contexts. Ideally the first contract can be locked down when upgrading, such that no data can be added, but often this is not necessary. That is because the first contract is simply a fall-back, and only in the case that no *new* data exists (which would be stored in the second contract) is the first ever called.

There are also some drawbacks to be aware of. If users continue using the first contract, they might be able to change the data returned from the second, newer contract. The approach may also require software updates depending on the dapp in question. Finally, this technique does not work when contracts need to return dynamic arrays or strings, as these cannot be passed between contracts. So whether this technique is appropriate depends on the nature of the contract being upgraded.

SecureVote's delegation contract was designed to handle all delegation requirements across all democracies, so voters could simultaneously delegate on a per-token basis and globally. If a delegation for a particular token was not found, the global delegation would be used. This is useful for self-delegating from a cold wallet to a hot wallet. SecureVote's first improvement was logging all known tokens

for which delegations had been made, allowing them to iterate through all known tokens easily.

SecureVote's other improvement was to look up all delegations to a particular address, exposed in a function called `findPossibleDelegatorsOf`. This was done by looping through all known delegations and constructing an in-memory array of delegations matching the delegate in question. One consequence of this approach is that only *potential* delegators are returned; delegations need to be checked individually before being treated as valid when calculating the results of a ballot. This demonstrates a trade-off between work done on-chain and work done off-chain. If backlinks were stored with the delegations themselves, the contract would also require additional data structures and logic to store and maintain the accuracy of these backlinks, increasing the cost for the voter. However, the chosen approach implies that the `findPossibleDelegatorsOf` function cannot be called from other smart contracts. When this function is called, the computation is only ever done on the Ethereum node responding to the call, not across all full nodes on the Ethereum network itself.

Complex Upgrades

The method above may be applicable for individual contracts but does not support upgrades of a complex system of contracts. The pattern used by SecureVote in Listing 13.5 (called an 'upgrade pointer') is suitable for singly linked contracts, where the contract being called might be upgraded. In this example `AContract` would call `checkIndexForUpgrade()` before sending any data to `Index`.

This example code shows the core idea, but the `doUpgrade` function could be easily extended to allow for upgrade hooks or notifications to be sent to other contracts. SecureVote extensively use such an extension to manage the multiple interactions and permissions between Tokenvote's smart contracts. A sample from their `Index` contract is shown in Listing 13.6.

Multiple other contracts are notified of an upgrade via their `upgradeMe(address)` method. SecureVote use this mostly for permission management, but it supports other complex upgrades. When `Index` calls `upgradeMe` on another contract, the permissions of `Index` are transferred to the new contract. Note also the modifiers `only_owner`, which allows only the owner of the contract to execute this function, and `not_upgraded`, which checks that the function can only be called on the latest version of the `Index`.

Atomic Upgrades and Tokenvote

Two lines in Listing 13.6 differ from the others: `ensOwnerPx.setAddr(nextSC);` and `ensOwnerPx.upgradeMeAdmin(nextSC);`. The contract `ensOwnerPx` is an 'owner proxy' for `index.tokenvote.eth`, defined using the Ethereum Name Service. Since this contract has administrative control over this

```
contract Upgradable {
  address public _upgradePtr;
}

contract Index is Upgradable {
  function doUpgrade(address next) external {
    require(msg.sender == owner);
    _upgradePtr = next;
  }
}

contract AContract {
  Index index;

  function checkIndexForUpgrade() internal {
    if (index._upgradePtr() != address(0))
      index = Index(index._upgradePtr);
  }
}
```

Listing 13.5 This pattern allows other smart contracts to know about an upgrade and act accordingly

```
function doUpgrade(address nextSC) only_owner() not_upgraded
    () external {
  doUpgradeInternal(nextSC);
  backend.upgradeMe(nextSC);
  payments.upgradeMe(nextSC);
  ensOwnerPx.setAddr(nextSC);
  ensOwnerPx.upgradeMeAdmin(nextSC);
  commAuction.upgradeMe(nextSC);

  for (uint i = 0; i < bbFarms.length; i++) {
    bbFarms[i].upgradeMe(nextSC);
  }
}
```

Listing 13.6 A sample from SecureVote's Index contract showing how upgrades notify other linked contracts. © 2018 SecureVote, reprinted with permission

name, it can expose functionality (like setting the address associated with the ENS name) to multiple other accounts. In this case, the other accounts are a SecureVote cold wallet and the `Index` contract.

When other smart contracts interact with Tokenvote, they first resolve the ENS name to an address to ensure they are invoking the current version of Tokenvote and not an old contract. Somewhat similarly, when voters use the Tokenvote UI, the software finds the `Index` address via an ENS lookup, but this only guarantees the address is correct at the time of the lookup. As mentioned above, there may be a race

condition if an upgrade is made after the user loads the UI but before they create a new ballot.[16]

As an aside, SecureVote have made extensive use of events in their contracts, though unfortunately forgot to add an event for upgrades. If they had included one, it would be entirely feasible for the UI to listen for upgrade events and to reload contract instances when such an event is emitted. This would reduce the risk of race conditions causing failed transactions.

13.4.4 Reducing Complexity and Cost

In preparation for production deployment, SecureVote began benchmarking and optimizing many of the methods users would call regularly, like casting a vote or creating a ballot. Several optimizations greatly reduced transaction fees for the end user. One advantage of platforms like Ethereum is that the performance and gas cost of function invocation can be measured accurately in test environments. As mentioned in Chapter 9, rather than implementing dynamic gas costs, Ethereum's design opts for a dynamic price per gas operation. Thus measuring (and optimizing) gas costs is separate from the price of a transaction.

During SecureVote's benchmarking, the primary offender in terms of gas use was identified to be the creation of a new ballot. The original architecture used a contract factory to deploy individual contracts to manage each ballot. Although this allowed them to reuse much of the code from the MVP, as they added features the cost of deployment grew to 3,000,000 gas. During the worst periods of congestion,[17] this corresponded to a transaction fee of around US$30. SecureVote felt this was too high and designed a more sustainable architecture for ballot creation.

A standard solution to this kind of problem is to refrain from deploying new contracts. Instead of deploying one contract per ballot, SecureVote would deploy one contract per *type of ballot*: a ballot storage contract, implementing the data contract pattern described in Section 7.4.2. In order to maximize code reuse, ballot-specific functionality like recording votes was not included in the ballot storage. Rather, this was refactored out into a library of its own, allowing SecureVote to reuse the ballot storage contract and interface with different libraries for vastly different kinds of ballots. The results of this new architecture reduced ballot creation cost to between 200,000 and 300,000 gas, a reduction of 85–95%.

However, this kind of change has many flow-on impacts. For example, their previous voting and auditing architecture included the assumption that each ballot lived at its own address. Under this new pattern, many ballots lived at the same

[16] Direct user interaction with the index only occurs on write operations; read operations call the backend directly, so an update to the index does not affect this functionality.

[17] Such as the CryptoKitties Congestion Crisis of late 2017. https://media.consensys.net/the-inside-story-of-the-cryptokitties-congestion-crisis-499b35d119cc.

address, and each had a unique identifier. Not only did voting-specific code require updating, but the entire state model of the UI required refactoring to accommodate ballot storage contracts, each holding multiple ballots. Even the URL routing logic needed to be updated.

In general a 'hub and spoke' architecture (where new spokes are created via newly deployed contracts) should only be used in cases where the cost of such deployment is warranted. Any developers using this architecture should strongly consider whether refactoring to a single, heavily used contract will improve performance, user experience, and maintainability.

13.5 Summary

In this chapter, SecureVote described their experience of moving from an MVP to a production dapp. We contrast the initial, planned architecture and the final one, which resulted from many lessons learned and optimizations made during the development. We also describe how the architecture relates to the functions blockchain can play, which patterns are used and how, as well as the considerations of the qualities and the trade-offs in the architecture. Finally, the previous section covers many details, code samples, and (occasionally hard) lessons learned.

13.6 Further Reading

The Tokenvote source code is available on GitHub at https://github.com/secure-vote/sv-light-smart-contracts.

Solidity documentation, including a very good 'by example' section, is available at https://solidity.readthedocs.io/en/latest/.

The end-to-end verifiable voting design *Prêt à Voter* is described in Ryan et al. (2009).

Chapter 14
Case Study: originChain

A Blockchain-Based Food Traceability System

with Qinghua Lu

14.1 Introduction and Background

A traceability system enables tracking products by providing relevant information (e.g. origination, item status, events, or locations) during production and distribution. Product suppliers and retailers often work with independent traceability companies who are certified to inspect the products throughout the supply chain. If everything satisfies the supply chain quality requirements, the traceability company issues inspection certificates that attest to the quality and origination of products. A traceability system is employed to expose these assurances as certificates. In this context, security is important for accountability and auditability. A traceability system normally stores information in conventional databases controlled by the company. However, such a centralized data storage becomes a potential single point of failure (operationally, and as a business) and is exposed to the risk of insider tampering.

originChain is the name of the blockchain system concept being developed by an independent third-party traceability provider (called 'ITTP' in this chapter). ITTP's conventional system provides traceability information for products imported from overseas to China. This system has been integrated with several large e-commerce websites in China and with public service agencies. Its traceability services are used by hundreds of product suppliers and retailers to manage traceability information for their products, and by millions of product consumers to access traceability information. Each product supplier, on average, has about 20 products to be traced; the granularity of the traceability information is rather coarse as it corresponds to product packages rather than individual products.

X. Xu et al., *Architecture for Blockchain Applications*,
https://doi.org/10.1007/978-3-030-03035-3_14

This chapter discusses a pilot study on replacing the conventional traceability system with one based on blockchain. For this purpose, we first extracted a traceability process with scenarios adapted from ITTP's current system. Based on this process, we created a new architecture, replacing the central database with blockchain and designing the architecture to support better adaptability. The new system can provide transparent tamper-proof traceability information, enhance the availability of the data, and automate regulatory compliance checking. We describe the implementation of this new architecture and its tests under realistic conditions, using data from the conventional system.

To assess the suitability of blockchain—see also Section 6.1—the following factors were taken into account. originChain is a multiparty system that spans many participants such as product suppliers, traceability companies, and service providers such as testing labs. These participants create and update information used in product traceability and also access information recorded by others. Data transparency and immutability are desired because participants and consumers need to check the origin and authenticity of the products. So, blockchain may be an appropriate technology to use. Because of the nature of the traceability information and business characteristics such as the numbers of suppliers and products, the performance of blockchain is likely to be adequate in this use case.

14.1.1 Traceability Process

In order to illustrate the use case, we depict a simplified traceability process as shown in Fig. 14.1, using BPMN.[1] The traceability company here would administer originChain, coordinate multiple service providers to perform inspection services, and issue certificates based on information provided by the service providers.

The process starts when a product supplier submits a quality tracing application for a batch of products to the traceability company. The administrator processes the application paper work (e.g. trading contracts, invoices, and order forms) and payment. Every batch of products triggers the application of traceability services. The agency assigns a factory examiner to inspect the factory at its address, including the factory's production capability and quality control process. After inspecting the factory, a freight yard examiner is sent to examine products in freight yards and to inspect on-site loading. The examiner attaches lead seals to the containers with products if the process of on-site loading meets requirements. Meanwhile, a product sample is sent to labs for testing. Once the application passes inspections and testing, the traceability company issues the supplier a traceability certificate for the commodity. Traditionally, all the traceability-relevant information and certificates are stored in a conventional database maintained by the traceability company.

[1] Business Process Model and Notation—http://www.bpmn.org/.

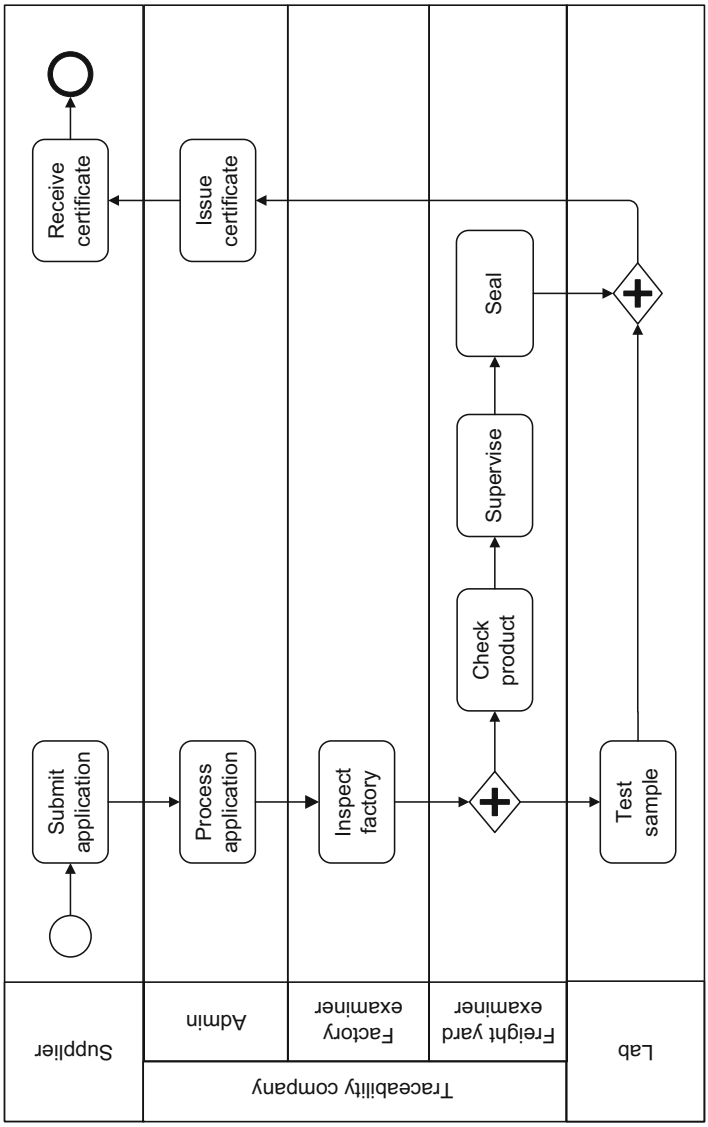


Fig. 14.1 Simplified originChain traceability process (notation: BPMN). © 2017 IEEE. Reprinted, with permission, from Lu and Xu (2017)

14.2 Architecture of originChain

A blockchain architecture for originChain is illustrated in Fig. 14.2. It consists of a UI layer, management layer, data layer (off-chain), and blockchain layer containing both data and some business logic. In terms of the categories in Chapter 5, the blockchain layer provides storage, computation, and communication.

14.2.1 Users of originChain

There are three types of users: service users, traceability company/service providers, and blockchain administrators. Service users include product suppliers, retailers, and consumers. Product suppliers manage product and enterprise information through a *Product & Enterprise Management* module, while product retailers and consumers check the quality and origin of products through a frontend in the originChain system. Suppliers apply for traceability services through the system and aim to receive certificates for compliance with traceability regulations, to demonstrate the quality and origin of their products to retailers/consumers. Retailers and consumers trust ITTP as a third-party, to verify the quality and origin of products.

Depending on agreements reached by the users and ITTP, services may include factory examination, sample testing, product checking, on-site loading supervision, and sealing. ITTP manages sample testing results through a *Sample Test Management* module and manages traceability information, certificates, and on-site photos using a *Traceability Management* module.

Blockchain administrators develop and deploy smart contracts in a *Smart Contract Management* module and control permissions of smart contracts using a *Permission Control* module. The settings of the blockchain network are managed using a *Blockchain Management* module.

14.2.2 On-Chain vs. Off-Chain

What should be stored on-chain and what should be stored off-chain is a major design issue for blockchain-based applications. The blockchain contains a full history of all transactions that have ever occurred in the blockchain network. Such information remains on the blockchain permanently. The ever-growing size of blockchain and full replication creates challenges for data storage on blockchains.

originChain only stores sensitive and small data on-chain. This includes the hashes of certificates and on-site freight yard photos, other traceability information, and permission control information. Traceability certificates and photos are critical for end users, and storing the hashes on blockchain can guarantee data integrity.

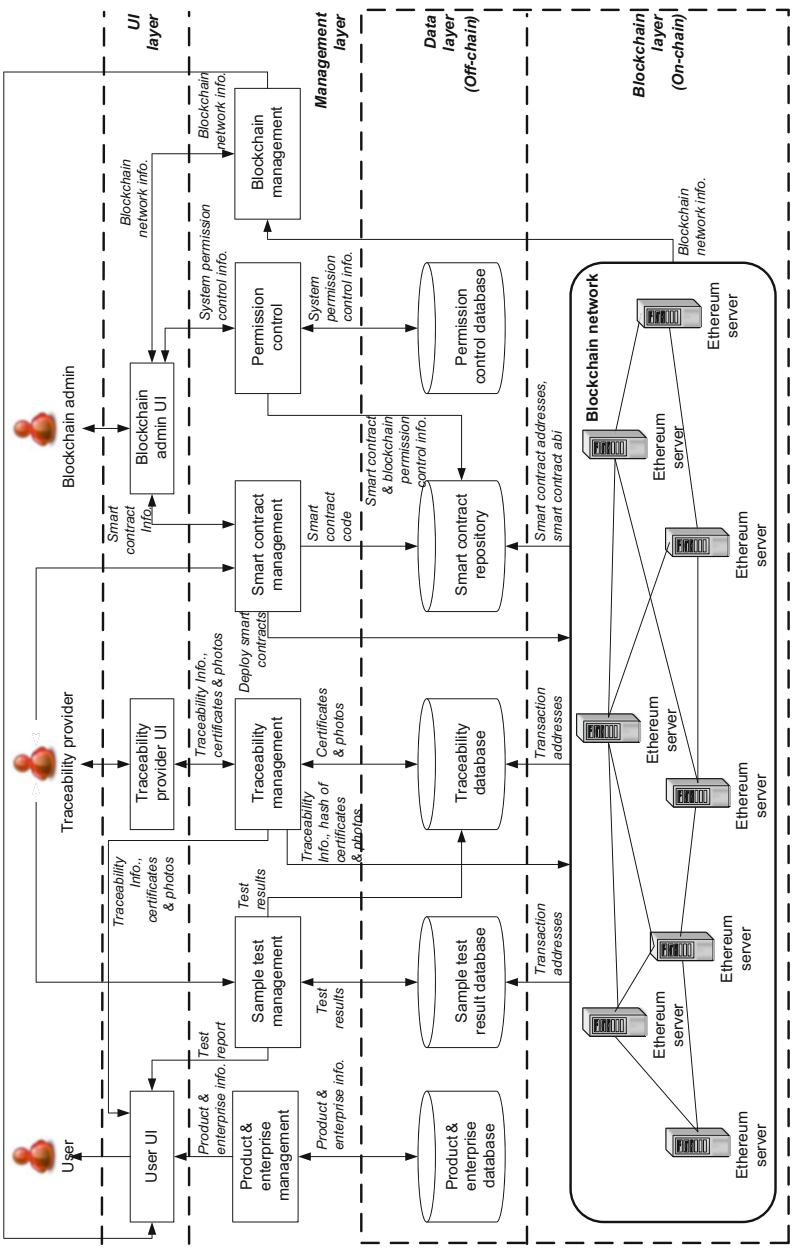


Fig. 14.2 Blockchain architecture for originChain

Blockchain transactions store this hash value as a proof of existence of the original raw file. Traceability information includes the information about product (batch number, traceability result and number, place of origin, manufacturer, manufacture date, and expiration date) and freight yard (loading port and inspection date). This information is stored on-chain as variables in smart contracts. Information about blockchain-level permission control is stored on-chain in a separate smart contract.

Off-chain data includes smart contract addresses, product/enterprise information, traceability certificates, photos, and smart contract source code. The information displayed for users consists of product-/enterprise-/traceability-related information and blockchain-related information. originChain saves the addresses of smart contracts off-chain to access data on-chain and stores the information of product/enterprise off-chain due to its size and non-sensitivity. Traceability certificates and photos are large and not suitable to store on-chain and therefore kept off-chain.

The smart contracts deployed on blockchain are in a binary format rather than source code. Thus the human-readable source code of smart contracts in high-level programming languages is stored off-chain in a *smart contract repository* for the blockchain administrators to manage.

14.2.3 Design of Smart Contracts

Blockchain can be used as a software connector, providing coordination services for components to coordinate through shared data and smart contracts. Figure 14.3 illustrates the high-level design of smart contracts used in originChain.

There is a *factory contract* deployed on the blockchain as a template to generate smart contracts corresponding to the agreements between ITTP, service providers, and product suppliers. The factory contract contains a list of contract templates, which represent different combinations of traceability services. The factory contract is called when the supplier submits the web form through the frontend UI. The conditions defined in the web form, e.g. what traceability services are selected, are passed to the factory contract through parameters for the factory contract to instantiate new smart contracts. Other than the conditions defined in the legal agreement, the generated smart contract also implements functions to check if all the information required by regulation is provided. Other regulatory rules and procedures can be implemented in the smart contract as well. A parametrized factory contract provides a flexible way to create smart contracts and improves the confidence that the smart contract is not modified by unauthorized people.

When the factory contract is called, a *registry contract*, a *service contract*, and a *data contract* are created. This design implements the patterns contract registry, data contract, and factory contract from Section 7.4.

The registry contract represents the legal agreement. The legal agreement is linked with the on-chain smart contract by adding the address of the smart contract into the legal agreement and adding the hash of the legal contract back to the

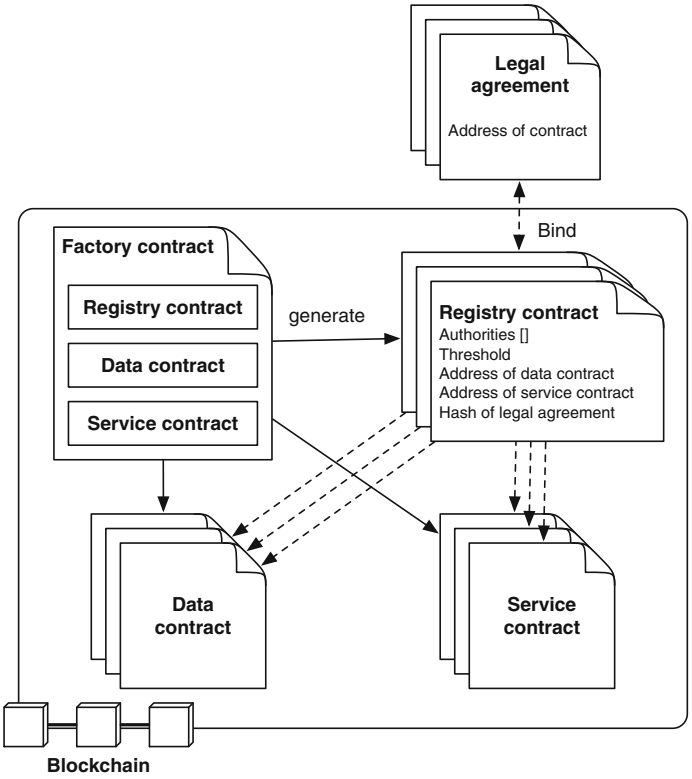


Fig. 14.3 Structural design of smart contracts

smart contract—see also the legal and smart contract pair pattern (Section 7.1.3). By binding a physical agreement with a smart contract, a bridge between the off-chain physical agreement and the on-chain smart contract is established. The smart contract codifies the conditions defined in the agreement. These conditions can be checked and enforced automatically by the smart contract. The smart contract also enables some automated regulatory compliance checking.

The registry contract contains the addresses of the service contract and the data contract. Separating data and control is a basic principle in software design. Such a separation allows the logic to be updated without affecting the data (and vice versa). The data contract is not supposed to change often, while the control contract can be much more flexible.

All the smart contracts running on blockchain can be accessed and called by all the blockchain participants by default, because there are no privileged users and every participant can join the network to access all the information and code on blockchain. A permission-less function might be triggered by unauthorized users. Empirical studies show that many smart contracts on the public Ethereum blockchain can be terminated without authority. To prevent unauthorized invoca-

tion, every smart contract in originChain has an embedded mechanism to check permissions for every caller that invokes any of the smart contract operations. As such, the embedded permission pattern (Section 7.4.3) is also implemented.

14.2.4 Dynamic Behaviour of Smart Contracts

In originChain, conducting an activity (i.e. processing a transaction) might be based on multiple authorities (i.e. multiple account addresses). Smart contracts are specified with a list of addresses that can authorize the invocation of certain functions. Quorums of a minimum number of addresses required to authorize a transaction can also be specified.

Here, an authority is a validation oracle that signs transactions based on external state. This might block the progress of a transaction until a condition over the external state is verified by the validation oracle who controls one or multiple of the predefined trusted addresses.

One reason to introduce multiple authorities is that blockchain does not offer mechanisms to recover a lost or a compromised private key. Losing a key results in permanent loss of control over an account or smart contract. Using the mechanism of multiple authorities, one participant can control more than one blockchain address, to reduce the risk of losing control over their smart contracts due to a lost or compromised private key. The list of the authority addresses can be also updated with authorization from a quorum of trusted addresses.

14.2.5 Permission Control and Blockchain Management

As shown in Fig. 14.2, the deployment of the blockchain layer is as a geographically-distributed consortium blockchain within ITTP, which has branch offices in three countries. The vision is to establish a trusted platform that covers other organizations, including labs certified by government, big suppliers, and retailers that have long-term relationship with the company (e.g. e-commerce companies that have already built trusted reputations with their customers).

The control of data on the blockchain is stored in a permission control smart contract that defines permission for content management, writing smart contracts and joining the consortium blockchain. To join the traceability platform, a company sends a request off-line. After successfully passing a number of checks, ITTP updates the permission control smart contract, which will allow the requesting company to join the blockchain network and synchronize the historical blockchain data.

14.3 Analysis

On the basis of the prototype implementation, a number of analyses were conducted to assess if the approach fulfilled requirements and expectations for originChain. Here we discuss specifically the topics adaptability and latency.

14.3.1 Qualitative Analysis: Adaptability

Changes handled by the originChain system include adding or removing traceability services from the legal agreement after the initial legal agreement is signed or dynamic binding of testing labs based on their availability. The structural design of smart contracts affects how easy it is to update smart contracts and thus the adaptability of the whole system.

As discussed in Section 14.1.1, traceability services can be dynamically defined, because the legal agreements signed between the product supplier/retailer and ITTP often change due to customizations of the traceability process. Customization is done through the factory contract. The function of a service contract can be updated by replacing the address of its old version with the address of a new version. As long as the interface between the service contract and the data contract is the same, the updated service contract can still use the previously stored data.

If there are new requirements, for example, new types of information required by a new regulation or new services provided by the service providers, the factory contract can be updated to a new version to fulfil the new requirements. In this case, the old factory is disabled, and the configuration of other modules is updated accordingly.

Dynamic binding of labs is enabled by multiple authorities. Multi-signature is a mechanism in Bitcoin that requires more than one private key to authorize a transaction. In Ethereum, we can use functions defined in smart contracts to implement a multi-signature mechanism. More flexibly, an M-of-N multi-signature can be used to define that M out of N public keys are required to authorize a transaction. We call M the quorum, or threshold of authority.

The *MultiSignature* contract in Listing 14.1 implements a multi-signature mechanism. In originChain, a user sends a request to originChain to issue a certificate. Such a request requires the approval from both the corresponding service provider and ITTP. The *IssueCert* contract inherits *MultiSignature* to use the mechanism.

The addresses of trusted authorities are predefined. The modifier *agreeSignature()* adds additional code to the function *issue()* to make sure that certain conditions are met before proceeding to execute the body of function. An authority invokes *issue()* to agree to the request. *agreeResult()* is called to check whether there is a quorum of signatures every time *issue()* is invoked. If so, the certificate can be issued. The requester can withdraw the request by invoking the function *cancelAgreeRequest()*.

```
contract MultiSignature{
  uint total;
  address[] authorities;
  uint agreeThreshold;
  address agreeRequester;
  mapping(address => bool) agreeState;
  ...
  modifier agreeSignature(){
    agreeState[msg.sender] = true;
    if(agreeResult()){_;}
  }
  function agreeResult() internal returns
  (bool signatureResult){
    uint k = 0;
    for(uint i = 0; i < total; i++){
      if(agreeState[authorities[i]] == true)
          k++;
    }
    if(k >= agreeThreshold)
      return true;
    else
      return false;
  }
  function cancelAgreeRequest(){
    if(msg.sender == agreeRequester)
      ...
  }
  ...
}

contract IssueCert is MultiSignature{
    //inherits MultiSignature
    string temID;
    bytes32 temCertHash;
    mapping(string => bytes32) certificate;
    function set(string ID, bytes32 certHash){
        temID = ID;
        temCertHash = certHash;
    }
    function issue() agreeSignature(){
    //below replaces "_;" in agreeSignature()
        certificate[temID] = temCertHash;
    }
    ...
}
```

Listing 14.1 MultiSignature contract

```
contract DynamicBinding{
  struct hashSecret{
    bytes32 hashKey;
    bool init;
    bool verified;
  }
  mapping (address => hashSecret) secret;
  //distinguish the struct initiated by
  //different address
  function initial(bytes32 key){
    hashSecret a = secret[msg.sender];
    if(a.init != true){
      a.hashKey = key;
      a.init = true;}
  }
  function changeKey(string oldKey, bytes32 newKey){
    hashSecret a = secret[msg.sender];
    if(a.init == true)
      if(a.hashKey == sha256(oldKey))
        a.hashKey = newKey;
}
  modifier verify(address initiator, string inputKey){
    hashSecret a = secret[initiator];
    if(a.verified == false) && a.hashKey == sha256(inputKey))
        {_; }
    a.verified = true;
    }
...
}

contract BindingLab is DynamicBinding{
    ...
    function sampleTest(address initiator, string key)
    verify(initiator, key){...}
    //passing the parameter to the modifier
}
```

Listing 14.2 DynamicBinding contract

The *DynamicBinding* contract in Listing 14.2 provides more flexible permission control, where a user can authorize the execution of the smart contract by using a hash secret initiated by originChain. The hash secret is generated off-chain and verified on-chain—see also the off-chain secret pattern in Section 7.3.2. Similar as above, the *BindingLab* contract inherits the *DynamicBinding* contract to use the mechanism. To initialize a hash secret, an ITTP employee invokes the *initial* function to link the hash key with his/her address so that only the employee has permission to change the hash secret using the *changeKey()* function after the secret is revealed. The hash secret is exchanged off-chain to the authorized labs. The secret key is verified by the modifier *verify()*. If the result is true, the lab providing the secret key proceeds with the *sampleTest()* function to upload the result of a sample

test. The smart contract requires that the secret is sent in plain form through a transaction to verify the hash secret. Thus, after *verify()* is invoked once, the secret is revealed. As mentioned above, large data is stored off-chain, and hash secrets can be used for both off-chain and on-chain permission control.

14.3.2 Quantitative Analysis: Latency of Writing and Reading

We conducted experiments to test the performance of originChain. As discussed in Section 14.2.2, the hashes of traceability certificates and photos are stored on-chain to support data integrity. Generating and storing the hash value on blockchain and querying and comparing hash values are the main operations used by originChain and the end users. Thus, we here describe the experiments that focus on testing the latency of writing and reading hash values with both the blockchain and a central database. ITTP currently maintains a central database at one of ITTP's branch offices (Section 14.2.5). Thus, we conducted three groups of experiments on a local database (for the branch office hosting the database), a remote database (for the other branch offices that access the database remotely), and a physically distributed consortium blockchain.

Our experiments were run on an Ethereum-based consortium blockchain. The *difficulty* of the blockchain was set to `0x4000`. On the public Ethereum blockchain, the difficulty dynamically adjusts so that blocks are generated every 12 s on average. On a consortium blockchain, difficulty can be configured according to the desired throughput of the system. In our experiment, the average block interval is 13.3 s, the maximum block interval was up to 58 s, and the minimum block interval was 1 s. For each experiment setup, we conducted an experiment that ran 200 times.

Table 14.1 shows some statistics for write latency, in milliseconds. For blockchain, we report two times: inclusion time and commit time. Inclusion time is the time spent for the transaction to be first included into a block on the blockchain. For commit time, we assume 12-block commit—refer to Section 11.6 for an in-depth discussion. The write latency of a remote database is higher than a local database due to additional network latency. The write latency of a blockchain is much higher than a remote database because it includes both network latency for the propagation of transactions and blocks and latency introduced by the consensus process.

Table 14.1 Latency of writing (ms)

	Blockchain		Database	
	Inclusion	Commitment	Local	Remote
Minimum	1348	72,870	1	418
First quartile	15,971	152,749	8	435
Median	25,494	176,332	10	439
Third quartile	35,666	204,159	11	446
Maximum	106,374	592,270	20	542
Average	29,453	187,938	10	441

Table 14.2 Latency of reading (ms)

	Blockchain	Database	
		Local	Remote
Minimum	8	1	422
First quartile	10	12	437
Median	11	15	443
Third quartile	13	17	449
Maximum	129	33	485
Average	17	15	444

In this prototype, a private instance of Ethereum with the default consensus mechanism was used, for convenience. However, in a production implementation, it would be more likely that an alternative consensus mechanism would be used. Nakamoto consensus and proof-of-work can be slow to commit transactions and only has probabilistic commit guarantees. Nonetheless, even using Ethereum with proof-of-work, the transaction write performance was acceptable for this use case.

Table 14.2 compares the read latency of the three configurations, in milliseconds. The read latency of the blockchain is comparable to the read latency of a local database, because reading a blockchain does not send transaction to the blockchain network and so can be served immediately. In comparison to remote reading, blockchain is indeed significantly faster.

14.4 Discussion

14.4.1 Architectural Design of Blockchain-Based Systems

Due to the unique properties of blockchain, there are design considerations that are specific to blockchain-based applications. However, because smart contracts are programs running on the blockchain, some of the usual architectural design principles are applicable to smart contracts. The structural design of the smart contracts has a large impact on cost if a public blockchain is used. The cost to deploy a smart contract depends on its size because the code is stored on the blockchain, which costs a data storage fee that is proportional to data size. Thus, more lines of code cost more money. A consortium blockchain does not have to use a token/currency; therefore per-transaction cost is not an issue for a consortium blockchain. However, blockchain size is still a design concern because the size of the ledger grows due to immutability and the full replication of the blockchain across all participants. The constraints on the size of transactions and blocks also restrict the complexity of smart contracts.

14.4.2 On-Chain vs. Off-Chain

What functionality and data to handle on-chain and what off-chain is a critical consideration when designing applications on blockchain. Performance and privacy factors need to be considered. Performance highly depends on the deployment of the blockchain. For example, a consortium blockchain can be configured to have much better performance than a public blockchain. In the case of originChain, due to the characteristics of the current systems (low write throughput because of the coarse granularity of traceability information), limited throughput of the blockchain is not the main concern. However, the data on blockchain is publicly accessible to all the participants of the blockchain network. As such, private data (e.g. customers' personal information) should not be stored on-chain. In the context of traceability, large-sized sensitive raw data (e.g. traceability certificates and photos) are required to be tamper-proof. Thus, only the hash of raw data is stored on-chain, while the corresponding raw data is placed off-chain in a secure database.

14.4.3 Adaptability of Blockchain-Based Systems

Adaptability is a quality attribute required by many industrial projects that are inherently dynamic. However, adaptability is rarely discussed in existing work for blockchain-based systems. We view a blockchain as one component of a larger distributed system. In originChain, we implement some of the business logic on-chain as smart contracts. Thus, the structural design of smart contracts also affects the upgradability of smart contracts and the adaptability of the whole system. However, if the blockchain is used for data storage only, not much can be done to affect adaptability of the whole system. Moving some logic to the blockchain as done on originChain can leverage the trustworthiness of blockchain as a computational platform.

14.5 Summary

Blockchain enables decentralization in new forms of distributed software architectures, where components can reach agreements on the historical system states without trusting a central integration point. In this chapter, we described experiences from designing, implementing, and testing originChain, a blockchain-based product traceability system that restructures an existing system by replacing the central database with a consortium blockchain. Our experience shows that the design of smart contracts can improve the adaptability of the system. Our experiments demonstrated that using blockchain only negatively affected write operations because of

the consensus process, while there was positive impact on read operations for remote offices, because every participant hosts a local full copy of the blockchain data structures.

This chapter is partly based on our earlier works (Lu and Xu 2017).

Epilogue

Our goal with this book is to help software architects and engineers (and students and researchers in these fields) to understand the concepts and implications of using blockchain in their applications. We hope you benefited from reading it.

Here we first *summarize* some of the main points in the book. First, blockchain is just one of a number of elements in the architecture of an application—unless your application is a blockchain platform, of course. Blockchain can be used as a data store, a computing platform, a communication mechanism, and a vault for digital assets—see Chapter 5. Other elements in a typical system include user interfaces, cryptographic keys, software clients, enterprise systems and external services, and auxiliary databases. Before starting to design and develop, you should ask yourself if it is really necessary or advisable to use blockchain—see Chapter 6.

Blockchain has some rather unique properties, leading to specific trade-offs. The most prevalent of the *trade-offs is transparency vs. confidentiality*—however, the choices typically depend on the use case and the philosophy of the organization (or group of open-source developers) that is building the application.

Other main trade-offs concern *on-chain vs. off-chain*: which data, computation, and communication should be done through blockchain? These decisions can impact most system properties, including transparency/confidentiality, cost, performance, maintainability/upgradability, and availability.

Some trade-offs can be resolved by using clever design solutions, and we discussed 15 reusable patterns for such solutions in Chapter 7. Given the specifics of blockchain and smart contract development, it is also worth considering a model-driven engineering methodology, like the ones we discussed in Chapter 8. Models not only allow code generation but also cost estimation (Chapter 9) and performance simulation (Chapter 10).

Blockchain poses a number of challenges for dependability—see Chapter 11. For instance, the network could decide to lower its block gas limit to counter a DDoS attack; but the lower limit might prevent applications and legitimate users from deploying new smart contracts. Also, blockchains do not normally have built-

X. Xu et al., *Architecture for Blockchain Applications*,
https://doi.org/10.1007/978-3-030-03035-3

in support for transaction retry or abort, so you might want to implement those to improve overall system dependability.

The case studies highlighted how blockchain can be used to reduce counterparty risks in agri-supply chains, for voting, and for provenance tracking of food. One big lesson is that the immutability of deployed smart contracts means that you need to plan ahead if you want to be able to fix bugs, patch vulnerabilities, or add new features in the evolution of your blockchain-based application.

The big question now is of course: *what will the future hold for blockchain?* It is clear that blockchain technology will be transformative, but not exactly how. As we argued in the introduction, blockchain has the potential to change the fabric that connects people, companies, governments, and whole societies. To quote *Amara's law*:

> 'We tend to overestimate the effect of a technology in the short run and underestimate the effect in the long run.'

Blockchain networks can mirror complex networks, such as in supply chains, and can dynamically adjust to accept new participants. Decentralization can facilitate trade. For example, in decentralized energy networks, neighbours could trade locally produced and stored electricity. There are some natural areas of application where production use is already happening or will soon start. Many more applications and industries will follow. Business models will be disrupted, and new ones will be developed. However, many of the fundamental changes will only happen in the invisible infrastructure at the backend of applications, and users will not directly see how these technology changes lead to impact for them.

The current phase of blockchain technology development is characterized by a broad front of innovation, leading to a lot of diversity. There are over 1300 cryptocurrencies at the time of writing, and many blockchain platforms exist. Chapters 3, 5, and 6 can help readers navigate the space of technologies and facilitate decision-making. Following this phase of increased diversity, there might be a phase of consolidation, and a small number of heavily used platforms might emerge.

Even without the complexity of the diversity of platforms, many people are struggling with the concepts and the implications of blockchain. For people starting to work in the space, there is a steep learning curve, but because of the ongoing rapid rate of innovation, you will find that a year later you will still learn something new and important about blockchain every week. This can be daunting and requires good information and education, and we hope to contribute with this book by sharing our learnings and insights gained from working in the area for the past 3 years.

In the platform game, everybody wants to own the platform—every company wants to build the next app store. In blockchain, nobody needs to 'own' the technology platform; it can be democratized. However, second-level platforms can

be built on top of main chains, which in turn might be controlled or dominated by one organization.

What is clear is that we live in interesting times. We hope you enjoyed the book and are ready to be part of shaping the future of blockchain.

References

Alpern B, Schneider FB (1985) Defining liveness. Inf. Process. Lett. 21(4):181–185

Anderson R (2008) Security engineering, 2nd edn. Wiley, Hoboken

Anderson L, Holz R, Ponomarev A, Rimba P, Weber I (2016) New kids on the block: an analysis of modern blockchains. CoRR abs/1606.06530. http://arxiv.org/abs/1606.06530

Androulaki E, Barger A, Bortnikov V, Cachin C, Christidis K, De Caro A, Enyeart D, Ferris C, Laventman G, Manevich Y et al (2018) Hyperledger Fabric: a distributed operating system for permissioned blockchains. In: Proceedings of the thirteenth EuroSys conference. ACM, New York, p 30

Antonopoulos AM (2015) Mastering Bitcoin: unlocking digital cryptocurrencies. O'Reilly, Sebastopol

Asghar MR, Ion M, Russello G, Crispo B (2012) Securing data provenance in the cloud. In: Proceedings of the 2011 IFIP WG 11.4 international conference on open problems in network security, iNetSec'11. Springer, Berlin, pp 145–160. https://doi.org/10.1007/978-3-642-27585-2_12

Avizienis A, Laprie JC, Randell B, Landwehr C (2014) Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Secure Comput. 1(1):11–13

Azaria A, Ekblaw A, Vieira T, Lippman A (2016) MedRec: using blockchain for medical data access and permission management. In: 2016 2nd international conference on open and big data (OBD2016), Vienna, pp 25–30. https://doi.org/10.1109/OBD.2016.11

Back A, Corallo M, Dashjr L, Friedenbach M, Maxwell G, Miller A, Poelstra A, Timón J, Wuille P (2014) Enabling blockchain innovations with pegged sidechains. http://cs.umd.edu/projects/coinscope/coinscope.pdf

Bartoletti M, Pompianu L (2017) An empirical analysis of smart contracts: platforms, applications, and design patterns. ArXiv e-prints 1703.06322

Bashir I (2018) Mastering blockchain: distributed ledger technology, decentralization, and smart contracts explained, 2nd edn. Packt Publishing Ltd., Birmingham

Bass L, Clements P, Kazman R (2012) Software architecture in practice, 3rd edn. Addison-Wesley Professional, Boston

Beck K, Cunningham W (1987) Using pattern languages for object oriented programs. In: Conference on object-oriented programming, systems, languages, and applications (OOPSLA). ACM, Orlando

Becker S, Koziolek H, Reussner R (2009) The Palladio component model for model-driven performance prediction. J. Syst. Softw. 82(1):3–22

Bolch G, Greiner S, de Meer H, Trivedi KS (2006) Queueing networks and Markov chains: modeling and performance evaluation with computer science applications. Wiley, Hoboken

X. Xu et al., *Architecture for Blockchain Applications*,
https://doi.org/10.1007/978-3-030-03035-3

Bonneau J, Miller A, Clark J, Narayanan A, Kroll JA, Felten EW (2015) SoK: research perspectives and challenges for Bitcoin and cryptocurrencies. In: The 36th IEEE symposium on security and privacy (SP2015), pp 104–121

Brunnert A, van Hoorn A, Willnecker F, Danciu A, Hasselbring W, Heger C, Herbst N, Jamshidi P, Jung R, von Kistowski J et al (2015) Performance-oriented devops: a research agenda. arXiv preprint arXiv:150804752

Buterin V (2015) On public and private blockchains. https://blog.ethereum.org/2015/08/07/on-public-and-private-blockchains/

Carminati B, Ferrari E, Tran NH (2014) Secure web service composition with untrusted broker. In: 2014 IEEE ICWS. IEEE, New York, pp 137–144

Castro M, Liskov B (1999) Practical byzantine fault tolerance. In: Proceedings of OSDI, pp 173–186

Clack CD, Bakshi VA, Braine L (2016a) Smart Contract Templates: essential requirements and design options. 1612.04496

Clack CD, Bakshi VA, Braine L (2016b) Smart Contract Templates: foundations, design landscape and research directions. 1608.00771

Clark DD, Wilson DR (1987) A comparison of commercial and military computer security policies. In: 1987 IEEE symposium on security and privacy, pp 184–194

Clements P, Bachman F, Bass L, Garlan D, Ivers J, Little R, Nord R, Stafford J (2003) Documenting software architectures: views and beyond. Addison-Wesley, Boston

Crain T, Gramoli V, Larrea M, Raynal M (2017) (leader/randomization/signature)-free byzantine consensus for consortium blockchains. arXiv abs/1702.03068. http://arxiv.org/abs/1702.03068

Danezis G, Meiklegohn S (2016) Centrally banked cryptocurrencies. In: 23rd annual network and distributed system security symposium (NDSS2016), CA

De Gooijer T, Jansen A, Koziolek H, Koziolek A (2012) An industrial case study of performance and cost design space exploration. In: Proceedings of the 3rd ACM/SPEC international conference on performance engineering. ACM, New York, pp 205–216

Decker C, Wattenhofer R (2013) Information propagation in the Bitcoin network. In: Proceedings of the IEEE conference Peer-to-Peer networks (P2P)

Decker G, Weske M (2011) Interaction-centric modeling of process choreographies. Inf. Syst. 36(2):292–312. https://doi.org/10.1016/j.is.2010.06.005

Dimitriou T, Karame G (2013) Privacy-friendly tasking and trading of energy in smart grids. In: The 28th annual ACM symposium on applied computing (SAC), pp 652–659

Downey P (2016) The characteristics of a register. https://gds.blog.gov.uk/2015/10/13/the-characteristics-of-a-register/

Eberhardt J, Tai S (2017) On or off the blockchain? Insights on off-chaining computation and data. In: ESOCC 2017: European conference on service-oriented and cloud computing. Springer International Publishing, Oslo, pp 3–15

Ether (2016) Ether stats. https://ethstats.net

Eyal I, Sirer EG (2018) Majority is not enough: Bitcoin mining is vulnerable. Commun. ACM 61(7):95–102

Eyal I, Gencer AE, Sirer EG, van Renesse R (2016) Bitcoin-NG: a scalable blockchain protocol. In: USENIX NSDI, Santa Clara, CA

Fdhila W, Rinderle-Ma S, Knuplesch D, Reichert M (2015) Change and compliance in collaborative processes. In: IEEE international conference on services computing (SCC), pp 162–169. https://doi.org/10.1109/SCC.2015.31

Flynn BB, Huo B, Zhao X (2010) The impact of supply chain integration on performance: a contingency and configuration approach. J. Oper. Manage. 28(1):58–71

Fox A, Brewer EA (1999) Harvest, yield, and scalable tolerant systems. In: The 7th IEEE workshop on hot topics on operating systems, pp 174–178

Franks G, Al-Omari T, Woodside M, Das O, Derisavi S (2009) Enhanced modeling and solution of layered queueing networks. IEEE Trans. Softw. Eng. 35(2):148–161

Friedman M (1991) The island of stone money. Working papers in Economics, no. E-91-3, Hoover Institution, Stanford University, CA

García-Bañuelos L, Ponomarev A, Dumas M, Weber I (2017) Optimized execution of business processes on blockchain. In: BPM'17: International conference on business process management, Barcelona

Gervais A, Karame GO, Wüst K, Glykantzis V, Ritzdorf H, Čapkun S (2016) On the security and performance of proof of work blockchains. In: Conference on computer and communications security, Vienna

Gifford DK (1979) Weighted voting for replicated data. In: Proceedings of the seventh ACM symposium on operating systems principles. ACM Press, New York, pp 150–162

Gorton I, Klein J, Nurgaliev A (2015) Architecture knowledge for evaluating scalable databases. In: Working IEEE/IFIP conference on software architecture (WICSA), Montréal

Grigg I (2004) The Ricardian contract. In: The 1st IEEE international workshop on electronic contracting (WEC2004). IEEE, San Diego, pp 25–31

Hull R, Batra VS, Chen YM, Deutsch A, Heath III FFT, Vianu V (2016) Towards a shared ledger business collaboration language based on data-aware processes. In: ICSOC, the international conference on service-oriented computing. Springer, New York. https://doi.org/10.1007/978-3-319-46295-0_2

IBM (2015) Device democracy – saving the future of the internet of things. https://www-935.ibm.com/services/multimedia/GBE03620USEN.pdf

Idelberger F, Governatori G, Riveret R, Sartor G (2016) Evaluation of logic-based smart contracts for blockchain systems. In: Rule technologies. Research, tools, and applications. Springer, New York, pp 167–183

Ion I, Sachdeva N, Kumaraguru P, Čapkun S (2011) Home is safer than the cloud!: privacy concerns for consumer cloud storage. In: Proceedings of the seventh symposium on usable privacy and security, SOUPS '11. ACM, New York, pp 13:1–13:20. https://doi.org/10.1145/2078827.2078845

Iosup A, Yigitbasi N, Epema D (2011) On the performance variability of production cloud services. In: Proceedings of the IEEE/ACM international symposium on cluster, cloud and grid computing (CCGrid), pp 104–113. https://doi.org/10.1109/CCGrid.2011.22

Juskalian R (2018) Inside the Jordan refugee camp that runs on blockchain. MIT Technology Reviews. https://www.technologyreview.com/s/610806/inside-the-jordan-refugee-camp-that-runs-on-blockchain/

Kemme B, Alonso G (2010) Database replication: a tale of research across communities. In: Proceedings of the VLDB endowment, vol 3(1–2), pp 5–12

Kounev S, Brosig F, Huber N (2014) The Descartes modeling language. Dept of Computer Science, University of Wuerzburg, Tech Rep

Koziolek A, Koziolek H, Reussner R (2011) PerOpteryx: automated application of tactics in multi-objective software architecture optimization. In: Proceedings of the joint ACM SIGSOFT conference–QoSA and ISARCS, ACM, pp 33–42

Lamport L (1977) Proving the correctness of multiprocess programs. IEEE Trans. Softw. Eng. 3(2):125–143. https://doi.org/10.1109/TSE.1977.229904

Lamport L (1998) The part-time parliament. ACM Trans. Comput. Syst. 16(2):133–169

Lamport L, Shostak R, Pease M (1982) The byzantine generals problem. ACM Trans. Program. Lang. Syst. 4(3):382–401

Li G, Muthusamy V, Jacobsen HA (2010) A distributed service-oriented architecture for business process execution. ACM Trans. Web 4(1):2

Lo SK, Xu X, Chiam YK, Lu Q (2017) Evaluating suitability of applying blockchain. In: The 22nd international conference on engineering of complex computer systems (ICECCS), Fukuoka

López-Pintado O, García-Bañuelos L, Dumas M, Weber I (2017) Caterpillar: a blockchain-based business process management system. In: BPM'17: international conference on business process management, Demo track, Barcelona

Lu Q, Xu X (2017) Adaptable blockchain-based systems: a case study for product traceability. IEEE Softw. 34(6):21–27

Luu L, Narayanan V, Zheng C, KBaweja, Gilbert S, Saxena P (2016) A secure sharding protocol for open blockchains. In: ACM SIGSAC conference on computer and communications security (CCS), Vienna

Malkhi D, Reiter M (1997) Byzantine quorum systems. In: Proceedings of the twenty-ninth annual ACM symposium on theory of computing, STOC '97, pp 569–578. https://doi.org/10.1145/258533.258650

Mehta NR, Medvidovic N, Phadke S (2000) Towards a taxonomy of software connectors. In: ICSE, the international conference on software engineering, pp 178–187

Mendling J, Hafner M (2008) From WS-CDL choreography to BPEL process orchestration. J. Enterp. Inf. Manage. 21(5):525–542

Meszaros G et al (1998) A pattern language for pattern writing. In: Pattern languages of program design, vol 3, pp 529–574

Miller A, Juels A, Shi E, Parno B, Katz J (2014) Permacoin: repurposing Bitcoin work for data preservation. In: IEEE symposium on security and privacy, San Jose

Molloy MK (1982) Performance analysis using stochastic Petri nets. IEEE Trans. Comput. 100(9):913–917

Mont MC, Tomasi L (2001) A distributed service, adaptive to trust assessment, based on peer-to-peer e-records replication and storage. In: IEEE workshop on future trends of distributed computing systems

Morisse M (2015) Cryptocurrencies and bitcoin: charting the research landscape. In: The 21st Americas conference on information systems (AMCIS2015)

Nakamoto S (2008) Bitcoin: A peer-to-peer electronic cash system. http://bitcoin.org/bitcoin.pdf

Narayanan S, Jayaraman V, Luo Y, Swaminathan JM (2011) The antecedents of process integration in business process outsourcing and its effect on firm performance. J. Oper. Manage. 29(1):3–16

Natoli C, Gramoli V (2016) The blockchain anomaly. In: IEEE international symposium on network computing and applications (NCA). IEEE, New York

NI of Standards and Technology, UD of Commerce (2012) Secure Hash Standard - SHS: Federal Information Processing Standards Publication 180-4. CreateSpace Independent Publishing Platform. http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf

Object Management Group (2010) BPMN 2.0 by Example. http://www.omg.org/spec/BPMN/. v1.0. Accessed 10 Mar 2016

Omohundro S (2014) Cryptocurrencies, smart contracts, and artificial intelligence. AI Matters 1(2):19–21. https://doi.org/10.1145/2685328.2685334

Panayides PM, Lun YV (2009) The impact of trust on innovativeness and supply chain performance. J. Prod. Econ. 122(1):35–46

Pérez JF, Casale G (2013) Assessing SLA compliance from Palladio component models. In: 2013 15th international symposium on symbolic and numeric algorithms for scientific computing (SYNASC). IEEE, New York, pp 409–416

Poon J, Dryja T (2016) The bitcoin lightning network: scalable off-chain instant payments. https://lightning.network/lightning-network-paper.pdf

Prisco G (2015) Bankymoon introduces Bitcoin payments to smart meters for power grids. Bitcoin Magazine. https://bitcoinmagazine.com/20139/bankymoon-introduces-bitcoin-payments-smart-meters-power-grids/

Prybila C, Schulte S, Hochreiner C, Weber I (2017) Runtime verification for business processes utilizing the Bitcoin blockchain. Futur. Gener. Comput. Syst. (FGCS). https://doi.org/10.1016/j.future.2017.08.024

Ratha D, Eigen-Zucchi C, Plaza S (2016) Migration and remittances factbook 2016, 3rd edn. Tech. rep., World Bank Publications

Reijers HA, Mansar SL (2005) Best practices in business process redesign: an overview and qualitative evaluation of successful redesign heuristics. Omega 33(4):283–306

Rimba P, Tran AB, Weber I, Staples M, Ponomarev A, Xu X (2017) Comparing blockchain and cloud services for business process execution. In: ICSA'17: IEEE international conference on software architecture, short paper, Gothenburg

Rimba P, Tran AB, Weber I, Staples M, Ponomarev A, Xu X (2018) Quantifying the cost of distrust: comparing blockchain and cloud services for business process execution. Inf. Syst. Front. https://doi.org/10.1007/s10796-018-9876-1

Rosenfeld M (2014) Analysis of hashrate-based double spending. arXiv preprint. http://arxiv.org/abs/1402.2009

Royal D, Rimba P, Staples M, Gilder S, Tran AB, Williams E, Ponomarev A, Weber I, Connor C, Lim N (2018) Making money smart: empowering NDIS participants with blockchain technology, CSIRO

Ryan PYA, Bismark D, Heather J, Schneider S, Xia Z (2009) Prêt à voter:a voter-verifiable voting system. IEEE Trans. Inf. Forensics Secur. 4(4):662–673. https://doi.org/10.1109/TIFS.2009.2033233

Schad J, Dittrich J, Quiané-Ruiz JA (2010) Runtime measurements in the cloud: observing, analyzing, and reducing variance. Proc. VLDB Endow. 3(1–2):460–471. https://doi.org/10.14778/1920841.1920902

Schmidt DC (2006) Guest editor's introduction: model-driven engineering. IEEE Comput. 39(2):25–31. https://doi.org/10.1109/MC.2006.58

Schneider FB (1990) Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Comput. Surv. 22(4):299–319

Sompolinsky Y, Zohar A (2013) Accelerating Bitcoin's transaction processing: fast money grows on trees, not chains. https://eprint.iacr.org/2013/881

Squicciarini A, Paci F, Bertino E (2008) Trust establishment in the formation of virtual organizations. In: ICDE Workshops. IEEE Computer Society, New York

Staples M, Chen S, Falamaki S, Ponomarev A, Rimba P, Weber ABTI, Xu X, Zhu J (2017) Risks and opportunities for systems using blockchain and smart contracts. Tech. rep., Data61 (CSIRO), Sydney

Swan M (2015) Blockchain: Blueprint for a new economy. O'Reilly, US

Swanson T (2015) Consensus-as-a-service: a brief report on the emergence of permissioned, distributed ledger systems. https://allquantor.at/blockchainbib/pdf/swanson2015consensus.pdf

Szabo N (1997) Formalizing and securing relationships on public networks. First Monday 2(9). https://doi.org/10.5210/fm.v2i9.548

The World Bank (2016) Remittance prices worldwide: making markets more transparent. https://remittanceprices.worldbank.org/en/countrycorridors/

Tran AB, Xu X, Weber I, Staples M, Rimba P (2017) Regerator: a registry generator for blockchain. In: CAiSE'17: international conference on advanced information systems engineering, forum track, tool demonstration

Tran AB, Lu Q, Weber I (2018) Lorikeet: a model-driven engineering tool for blockchain-based business process execution and asset management. In: BPM'18: international conference on business process management, Demo track, Sydney, NSW, Australia

Tschorsch F, Scheuermann B (2016) Bitcoin and beyond: a technical survey on decentralized digital currencies. IEEE Commun. Surv. Tutorials 18(3):464

Valenta L, Rowan B (2015) Blindcoin: Blinded, accountable mixes for Bitcoin. In: FC, San Juan, Puerto Rico

van der Aalst WMP, Weske M (2001) The P2P approach to interorganizational workflows. In: International conference on advanced information systems engineering, pp 140–156. https://doi.org/10.1007/3-540-45341-5_10

van der Aalst W, ter Hofstede A, Kiepuszewski B, Barros A (2003) Workflow patterns. Distrib. Parallel Databases 14(1):5–51. https://doi.org/10.1023/A:1022883727209

Viriyasitavat W, Martin A (2011) In the relation of workflow and trust characteristics, and requirements in service workflows. In: Informatics engineering and information science. Springer, New York, pp 492–506

Vukolić M (2015) The quest for scalable blockchain fabric: proof-of-work vs. BFT replication. In: IFIP WG 11.4 international conference on open problems in network security, iNetSec, Zurich

Walport M (2016) Distributed ledger technology: beyond block chain. Tech. rep., UK Government Chief Scientific Adviser

Weber I, Haller J, Mülle J (2008) Automated derivation of executable business processes from choreographies in virtual organizations. Int. J. Bus. Process. Integr. Manage. 3(2):85–95. https://doi.org/10.1504/IJBPIM.2008.020972

Weber I, Xu X, Riveret R, Governatori G, Ponomarev A, Mendling J (2016) Untrusted business process monitoring and execution using blockchain. In: International conference on business process management. Springer, Rio de Janeiro, pp 329–347

Weber I, Gramoli V, Staples M, Ponomarev A, Holz R, Tran A, Rimba P (2017) On availability for blockchain-based systems. In: SRDS'17: IEEE international symposium on reliable distributed systems. IEEE, Hong Kong, pp 64–73

Willnecker F, Brunnert A, Krcmar H (2014) Predicting energy consumption by extending the Palladio component model. In: SOSP14 symposium on software performance: joint descartes/kieker/palladio days 2014, p 177

Wood G (2015–2018) Ethereum: a secure decentralized generalised transaction ledger. All revisions: https://github.com/ethereum/yellowpaper; latest revision: https://ethereum.github.io/yellowpaper/paper.pdf

Xu J, Woodside M, Petriu D (2003) Performance analysis of a software design using the UML profile for schedulability, performance, and time. In: International conference on modelling techniques and tools for computer performance evaluation. Springer, New York, pp 291–307

Xu X, Pautasso C, Zhu L, Gramoli V, Ponomarev A, Tran AB, Chen S (2016) The blockchain as a software connector. In: The 13th working IEEE/IFIP conference on software architecture (WICSA), Venice

Xu X, Weber I, Staples M, Zhu L, Bosch J, Bass L, Pautasso C, Rimba P (2017) A taxonomy of blockchain-based systems for architecture design. In: ICSA2017. IEEE, Gothenburg, pp 243–252

Xu X, Pautasso C, Zhu L, Lu Q, Weber I (2018) A pattern language for blockchain-based applications. In: EuroPLoP'18: European conference on pattern languages of programs, Kloster Irsee

Yasaweerasinghelage R, Staples M, Weber I (2017a) Predicting latency of blockchain-based systems using architectural modelling and simulation. In: IEEE international conference on software architecture (ICSA). IEEE, New York

Yasaweerasinghelage R, Staples M, Weber I (2017b) Using architectural modelling and simulation to predict latency of blockchain-based systems. Tech. Rep. 201704, School of Computer Science & Engineering, University of New South Wales, Sydney. ftp://ftp.cse.unsw.edu.au/pub/doc/papers/UNSW/201704.pdf

Yu X, Xu X, Liu B (2017) Ethdrive: a peer-to-peer data storage with provenance. In: CAiSE'17: international conference on advanced information systems engineering, forum Track, tool demonstration

Zhang P, White J, Schmidt DC, Lenz G (2017) Applying software patterns to address interoperability in blockchain-based healthcare apps. arXiv e-prints. https://doi.org/1706.03700

Index

X. Xu et al., *Architecture for Blockchain Applications*,
https://doi.org/10.1007/978-3-030-03035-3