

SUPPLY CHAIN FINANCE THROUGH A BLOCKCHAIN LENS

by

Anadi Pandharkar

B.E., Ujjain Engineering College, 2016

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE in BUSINESS ADMINISTRATION

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Transportation and Logistics)

THE UNIVERSITY OF BRITISH COLUMBIA
(Vancouver)

October 2020

© Anadi Pandharkar, 2020

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

Supply chain finance through a Blockchain lens

submitted by **Anadi Pandharkar** in partial fulfillment of the requirements for

the degree of **Master of Science in Business Administration**

in

Transportation and Logistics

Examining Committee:

Harish Krishnan, Professor, Operations and Logistics Division, UBC

Supervisor

Prof. Victoria Lemieux, Archival Science and Blockchain@UBC, UBC

Supervisory Committee Member

Prof. Robin Lindsey, Operations and Logistics Division, UBC

Additional Examiner

Abstract

The world experienced unprecedented growth in international trade in the past few decades. In this resulting environment of global competition, several new companies have spawned and are spawning. Sellers are often under immense pressure by the market players to accept open-account trade terms, shipping goods before receiving payment, leaving them exposed to increased risk. This creates working capital challenges for firms, especially small and medium enterprises. To mitigate this supply chain risk, several supplier-led and buyer-led supply chain finance solutions are facilitated by banks and financial technology companies. However, because of the new Basel III regulation framework for banks and several other Supply Chain Finance (SCF) adoption barriers, like fraudulent activities, many firms are unable to reap SCF's full benefits. This study explains various SCF instruments, the key drivers in their growth and their adoption barriers.

This study then focuses on the novel blockchain technology and smart contracts by delving deep into their history, components, limitations, risks and use cases. Using the knowledge gathered in the process, a proof-of-concept blockchain and smart contract is developed using the Ethereum platform, which can facilitate normal business, purchase order finance, reverse factoring and reverse securitization. To test the smart contract, four use cases for each SCF instrument mentioned are demonstrated. A JavaScript-based unit test is done to test the smart contract's correct deployment and onboarding of actors along with their business and financing interactions – access controls to business documents, reverting malicious transactions and correct fund transfer.

As a result of various assumptions taken in the development process, the smart contract works only as a basic proof-of-concept and lacks robustness on the ground of scalability and security. As a result, a future model is laid out which will use various software and hardware oracles for

autonomous operations while using a complex system of a storage contract, a permanent contract which stores all the data, and logic contract, upgradable contract which can be changed any number of times, with a proxy contract making delegated “function” calls to logic contract to reduce the gas usage due to external function calls.

Lay Summary

This study aims to provide a comprehensive account of various supply chain finance (SCF) instruments, their key drivers and adoption barriers to their growth. It further discusses the novel blockchain technology and smart contracts by studying its various components, advantages, risks and limitations. A major focus is on developing a proof-of-concept smart contract on Ethereum platform to facilitate business, purchase order finance, reverse factoring and reverse securitization. The proof-of-concept is then tested by demonstrating the use-cases for each SCF instrument and a JavaScript-based test. In the end, various flaws in the proof-of-concept are identified and are used as stepping-stones to lay the foundation of a more secure and scalable solution for the future.

Preface

This dissertation is original, unpublished and independent work by author, Anadi Pandharkar.

Table of Contents

Abstract.....	iii
Lay Summary	v
Preface.....	vi
Table of Contents	vii
List of Figures.....	xi
List of Images	xiii
List of Abbreviations	xiv
Acknowledgements	xvi
Chapter 1: Introduction	1
Chapter 2: Supply Chain Finance	4
2.1 What is a Supply Chain?.....	4
2.2 Supply Chain Finance	5
2.2.1 Working Capital Management.....	6
2.2.2 Open-Account Trade.....	8
2.3 Types of Supply Chain Finance	9
2.3.1 Purchase Order based Finance	10
2.3.1.1 Pre-shipment Financing	11
2.3.1.2 Inventory Financing	11
2.3.2 Invoice based Financing	12
2.3.2.1 Supplier-led Architecture.....	12
2.3.2.2 Buyer-led Architecture.....	13

2.3.2.2.1	Dynamic Discounting.....	14
2.3.2.2.2	Reverse Factoring.....	15
2.3.2.2.3	Reverse Securitization.....	16
2.4	Key Drivers and Limitations of Supply Chain Finance.....	17
2.5	Where Does Blockchain Fit?	20
2.5.1	Overcoming SCF barriers	22
Chapter 3: Blockchain Technology		25
3.1	Introduction.....	25
3.2	Types of Blockchains.....	27
3.3	Components of Blockchain.....	29
3.4	How Does Blockchain Work?	36
3.5	Advantages and Limitations of Blockchain.....	37
3.5.1	Advantages.....	37
3.5.2	Disadvantages	38
Chapter 4: Smart Contracts		40
4.1	History.....	40
4.2	Definition and Operations.....	41
4.3	Research Models for Smart Contracts	46
4.4	Smart Contracts in Supply Chain Management.....	49
4.5	Choosing the Blockchain	51
4.6	Limitations of Smart Contracts.....	54
4.6.1	Contract Limitations	54
4.6.2	Blockchain Limitations.....	55

Chapter 5: Proof-of-Concept Smart Contract	57
5.1 Introduction.....	57
5.2 Supply Chain Finance Models on Blockchain.....	57
5.2.1 Purchase Order Financing.....	60
5.2.2 Buyer-led Invoice Financing or Reverse Factoring	62
5.2.3 Reverse Securitization	64
5.3 Pseudocode	68
5.3.1 RegisterCompany.sol.....	71
5.3.2 Contract Interfaces	73
5.3.3 SafeMath.sol	73
5.3.4 CompleteFinancingContract.sol.....	74
5.4 Use Cases	83
5.4.1 Use Case 1: No - Finance.....	83
5.4.2 Use Case 2: Purchase Order Finance	84
5.4.3 Use Case 3: Invoice Finance	85
5.4.4 Use Case 4: Reverse Securitization	85
5.5 Testing Smart Contract	86
Chapter 6: Future Work and Conclusion.....	89
6.1 Future Work	89
6.2 Conclusion	98
Bibliography	100
Appendices.....	110
Appendix A RegisterCompany.sol	110

Appendix B CompleteFinancingContract.sol	113
Appendix C RegisterCompanyInterface.sol	133
Appendix D CompleteFinancingContractInterface	134
Appendix E SafeMath.sol	138
Appendix F Contract Structure	139
Appendix G Use Cases	140
G.1 Use case 1	140
G.2 Use Case 2.....	143
G.3 Use Case 3.....	143
G.4 Use case 4	144
Appendix H CompleteFinancingTest.js.....	146

List of Figures

Figure 2.1 Supply Chain Complexity (adapted from: Lambert & Cooper, 2000).....	5
Figure 2.2 Growth in O/A vs L/C (Source: Global Supply Chain Finance Forum, 2015)	8
Figure 2.3 Trigger Events for Supply Chain Finance (adapted from: Lamoureux & Evans, 2011 and Bryant & Camerinelli, 2013).....	9
Figure 2.4 Reverse Factoring Procedure (constructed by author)	15
Figure 2.5 Reverse Securitization procedure (adapted from: Hofmann et al., 2017)	17
Figure 3.1 Bitcoin price vs Blockchain Google Search	26
Figure 3.2 Timeline to Current Day Blockchain (adapted from: Kaligotla and Macal (2018)) ...	27
Figure 3.3 Types of Blockchain.....	29
Figure 3.4 Merkle Tree Structure (constructed by author)	31
Figure 3.5 Working of Blockchain (constructed by author)	37
Figure 4.1 Blockchain Protocol Stack (adapted from: Rosa et al., 2019).....	42
Figure 4.2 Working of a smart contract (adapted from: Wang et al., 2019).....	44
Figure 4.3 Blockchain Research Model (adapted from: Wang et al., 2019)	46
Figure 5.1 Proof -of-Concept Onboarding Process (constructed by author)	59
Figure 5.2 Proof-of-Concept Purchase Order Financing (constructed by author).....	61
Figure 5.3 Proof-of-Concept Reverse Factoring (constructed by author)	63
Figure 5.4 Proof-of-Concept Reverse Securitization (constructed by author)	66
Figure 5.5 Smart Contract Composition (constructed by Author).....	70
Figure 5.6 Unit Test Results of the Smart Contract.....	87

Figure 6.1 Future Model for Blockchain based Purchase Order Financing (constructed by the author)	97
---	----

List of Images

Img G.1 Compilation Details Remix	140
Img G.2 Test Accounts in Ganache	140
Img G.3 Contract Creation.....	140
Img G.4 Onboarding Transactions.....	141
Img G.5 Company Details Queried from Blockchain	141
Img G.6 Product Details Queried from Blockchain.....	141
Img G.7 Supplier Successfully Queried Inventory	141
Img G.8 Purchase Order, Invoice, Bill of Lading and Shipment Successfully Queried.....	142
Img G.9 Supplier's REC Balance After Buyer Confirms Shipment	142
Img G.10 Successful Payment by the Buyer	142
Img G.11 Successful Ether Withdrawal by the Seller	142
Img G.12 Successful Querying of Purchase Order and Invoice	143
Img G.13 Supplier Recieves Ether after POF approval	143
Img G.14 Bank Successfully Withdraws Ether after Buyer Pays the Invoice.....	143
Img G.15 Invoice generated Accepting Invoice Finance.....	143
Img G.16 Account Balances and REC Balance after Bank approves Invoice Finance	144
Img G.17 Securities and SPV details after Buyer applies for Reverse Securitization.....	144
Img G.18 Note details after Investor Buys Securities.....	144
Img G.19 Both the Seller Withdraw Ethers from the Contract.....	145
Img G.20 Both the Investors Withdraw Ethers from the Contract	145

List of Abbreviations

ABI Application Binary Interface

AP Accounts Payable

API Application Programming Interface

AR Accounts Receivables

BFT Byzantine Fault Tolerance

BTC Bitcoin

C All the liquid cash available

C2C Cash to Cash Cycle

DApp Distributed Application

DAO Decentralized Autonomous Organization

DeFi Decentralized Finance

DIH Days of Inventory in Hand

DPO Days Payable Outstanding

DSO Days Sales Outstanding

ERP Enterprise Resource Planning

ECDSA Elliptical Curve Digital Signature Algorithm

ETH Ethereum

EOA Externally Owned Account

EVM Ethereum Virtual Machine

FCF Free Cash Flow

FLP Fischer, Lynch and Patterson

GDP Gross Domestic Product

I Inventory including raw material and work in progress

IoT Internet of Things

IPFS Inter Planetary File System

IDE Integrated Development Environment

KYC Know Your Customer

L/C Letter of Credit

PO Purchase Order

PoW Proof of Work

PoS Proof of Stake

PBFT Practical Byzantine Fault Tolerance

REC Receivables Token

RPA Receivables Purchase agreement

SEC Security Token

SCF Supply Chain Finance

SME Small and Medium-sized Enterprise

SHA Secure Hash Algorithm

SPV Special Purpose Vehicle

TCP/IP Transmission Control Protocol/Internet Protocol

UTXO Unspent Transaction Output

UDP User Datagram Protocol

WC Working Capital

XOF Extensible Output Function

Acknowledgements

Quite naively until now, I was looking at the Supply Chain Management as a mere system of logistics and value exchange, but I was wrong. And I would like to express my deepest thanks to my supervisor, Prof. Harish Krishnan for introducing me to the world of supply chain finance and its interaction with blockchain technology. I would also like to thank Prof. Victoria Lemieux for realizing my potential and accepting me into the UBC Blockchain cluster as a student of Blockchain Graduate Pathway. Working with her in several projects has not only equipped me with knowledge of one of the most cutting-edge technologies in the world but she has also taught me how to become a true leader and a good spokesperson for a multi-cultural team. I would also like to express my gratitude to Prof. Robin Lindsey and Prof. Trevor Heaven for teaching me steps for proper research, professionalism and good writing practices.

Having never before been outside of my home country, India, I was not prepared for such a cultural shock. I would like to thank my family for their constant moral and financial support. Fortunate enough to have created life-long friendships, I would like to thank my friends from India and my roommates in Vancouver for pushing me through the time of distress.

Travelling back along back at the memory lane, I recall all the moments of joy, self-realization and hardship in the past few years. I would like to thank UBC for providing me with the opportunity to grow in such a diverse and beautiful environment.

Chapter 1: Introduction

The last decade has witnessed unparalleled growth in world trade due to globalization and the internet boom. Small firms were able to use e-commerce avenues to expand their business horizons across the globe. Small firms with big demand often have “big buyers” who are massive multinational companies and thus the small firms lack negotiation power over them. This has led to the subsequent rise of Open Account trade techniques where the seller ships order with invoice having due days of payment ranging from 30 to 90 days. Traditional financing techniques like letters of credit or bank payment obligation suffered a setback because of the fierce competition and their considerably slow pace to provide working capital to sellers. Open account trade was able to outpace traditional techniques once banks started to provide supply chain financing options to buyers and sellers (Sommer & O’Kelly, 2017).

Supply chain finance aims to resolve cashflow, the net cash transferred into and out of the business, conflicts between buyers and sellers, who both have vested interest in extending their cash flow cycle to “look good” to their investors and sustain their business. Financial institutions acting as an intermediary provide quick cash availability to sellers at a discounted rate depending upon either the buyer’s risk profile or the seller’s risk profile. At the maturity of the invoice, the buyer makes the payment to the financial institution.

But supply chain finance has suffered adoption problem due to obstructions from demand-side, supply-side, technological and regulatory obstacles (Lamoureux & Evans, 2011). Aswin (2019) provides a thorough study of supply chain finance barriers and identifies similar problems in financial, technology, organizational policies and various other domains. The Basel III framework,

an international banking framework in response to the 2007-09 financial crisis, seems to make these problems worse by putting more pressure on banks for extra regulatory compliance. As a result, the administrative costs associated with Know Your Customer (KYC) is going up. The European financial services group Nordea, wrote that a big chunk of the global transactions is non-repetitive and highly fragmented (*Trade Finance Is Going Open Account*, 2018). Hence, the financial institutions find it even harder to justify their costs unless they achieve a significant transaction volume. On top of that, there have been numerous cases of frauds associated with supply chain finance involving usage of fake invoices or fake purchase order to deceive the financial institutions. The focus of Chapter 2 of this study is to better understand the various supply chain techniques and adoption barriers associated with them.

The study then switches the attention to blockchain technology and how its inherently distributed, and secure nature mitigate the problems with SCF adoption. Chapter 3 delves into various types of Blockchain and its components, where blockchain would fit as a solution to the problems in supply chain finance and its limitations.

Chapter 4 investigates smart contract, a tool for embedding business logic in blockchain, and how smart contracts could lead to the rise of a new model of supply chain financing. A proof-of-concept smart contract on the Ethereum blockchain platform is developed to demonstrate the capabilities and limitations of this technology. Chapter 5 provides a detailed explanation of the construction of smart contract using the supply chain finance models adopted in the process for normal business, purchase-order financing, reverse factoring and reverse securitization. It discusses a test based on a JavaScript library Chai.js and Mocha.js to test the smart contract on 10 different aspects.

Finally, Chapter 6 discusses the various security and scalability loopholes. Further, I propose a future model which involves the use of oracles on a different Quorum blockchain, to have a more secure and scalable solution.

Chapter 2: Supply Chain Finance

2.1 What is a Supply Chain?

The core element to the economy, in any context, is essentially supply and demand and how these two variables interact with each other. Every player sources raw materials, perform some operations on to produce goods and then ships them for the subsequent consumption, hence contributes to both supply and demand part of the economy. The term supply chain was first used on 4 June 1982 by Keith Oliver in an interview with Arnold Kransdorff of Financial Times (Karaffa, 2012). The term was slow to take hold but eventually the term ‘supply chain management’ became ubiquitous.

Often confused with logistics, which involves only moving of goods, supply chain management encompasses material, information and financial flow across different organizations. It deals with forecasting, inventory management and every other operation which is required to make the goods available to the end consumer from a supplier.

An example of a typical Supply Chain network is given in Figure 2.1. It consists of a multitude of layers of suppliers and downstream distributors and retailers. It also shows the flow of information, money and goods in the supply chain from tier 1 supplier to consumer, unless it's a recall – the direction reverses. With every additional tier, there is an additional layer of complexity and it generates a need to share information with the new entity. All the players in the network play a very specific role. Failure of even one entity, due to any typical supply disruption, like natural disaster or bankruptcy, can lead to a domino effect in the supply chain and the entire network suffers the cost. Not much can be done to prevent a natural calamity but financial blows like

bankruptcy can be avoided. For example, suppose a crucial supplier for a focal manufacturing company of a network collapses due to low working capital. The production will most likely stop, and the supply chain will suffer. To mitigate this and provide benefit to both supplier and the manufacturer, a supply chain finance model can be adopted.

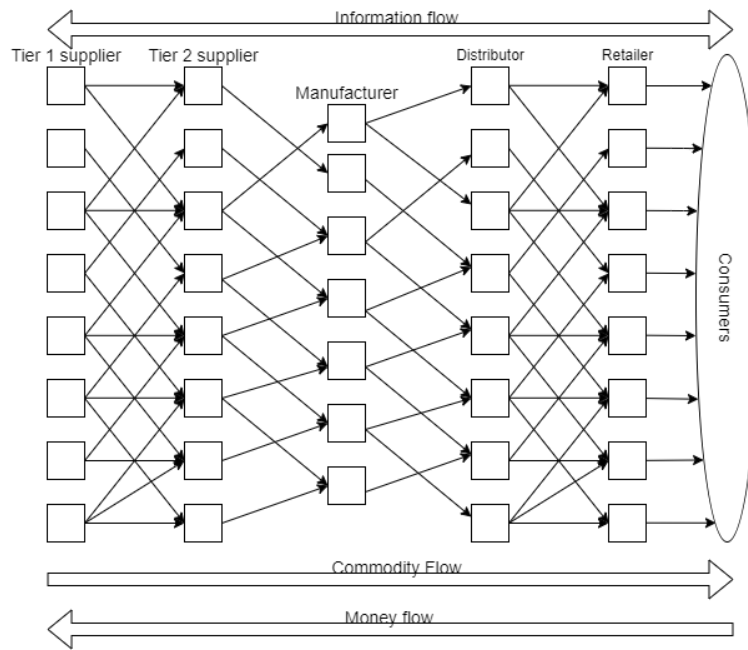


Figure 2.1 Supply Chain Complexity (adapted from: Lambert & Cooper, 2000)

2.2 Supply Chain Finance

Supply Chain Finance is a set of techniques employed by the bank to mitigate financial risk across the supply chain (Global Supply Chain Finance Forum, 2015). This was done to challenge the traditional way of financing the business using finance practices like letters of credits, bank guarantees etc. It can be more clearly understood from the definition provided by the Global Supply Chain Forum (Global Supply Chain Finance Forum, 2015):

“ The expression “supply chain finance” (SCF) today covers a wide range of products, programmes and solutions in the financing of commerce, including international trade, and has been used to refer to a single product, or a comprehensive range of products and programme of solutions aimed at addressing the needs of buyers and sellers, especially when trading on open account terms, in the increasingly complex supply chains in which they are involved. Visibility of underlying trade flows by the finance provider(s) is a necessary component of such financing arrangements usually enabled by a technology platform.”

To understand the above statement, a sound knowledge of working capital, open account trade and technology platform is required.

2.2.1 Working Capital Management

Working capital management of a business is dependent on managing two financial metrics: Cash to Cash Cycle and Free Cash Flow are required to be understood. Cash to Cash cycle accounts for the time-period between paying cash to supplier and receiving cash from customers and is a key metric for calculating the amount of cash required for operations. It can be calculated as follows:

$$C2C = DSO + DIH - DPO$$

Where C2C stands for Cash to Cash Cycle, DSO stands for Days Sales Outstanding which means an average number of days receivables remain outstanding before they are collected, DIH stands for Days of Inventory in Hand which means time in days in which inventory can be converted into cash and DPO stands for Days Payable Outstanding which means the number of days taken to pay

the invoices and bills. There is a conflict for C2C between buyer and seller because buyers want to increase their DPO and reduce DSO and vice versa for sellers.

Cash Flow is the net cash transferred into and out of the business and it decides business's ability to create value. It has three forms: operating, investing and financing. To measure how profitable business is, in terms of its expansion capability and return to the shareholder, free cash flow (FCF) is calculated as follows:

$$FCF = \text{Operating Cash Flow} - \text{Capital Expenditure}$$

Now, working capital (WC) depicts the daily operational liquidity in business to sustain its operation. It is given by

$$WC = AR + I + C - AP$$

Where 'AR' stands for Accounts Receivables, 'I' stands for inventory including raw material and work in progress, 'C' stands for all the 'liquid cash available and 'AP' stands for Accounts Payable.

By shortening its C2C cycle and increasing FCF, working capital (WC) can be optimized and the ability to self-sustain and expansion increases for business. But since every player in the chain holds the incentive to reduce C2C and increase FCF, tension grows. SCF can reduce this conflict by using a third party to intermediate and act as a financial buffer which can pay suppliers early and can bear the risk of late payment of buyer.

2.2.2 Open-Account Trade

Traditional forms of trade finance like Letter of Credit etc. were deemed to be perennial in trade. But the advent of the “e-commerce” revolution challenged this idea. Buyers and sellers, or importers and exporters, now had a variety of options. One of these options, open account trade, provides similar financing benefits to L/C with more flexibility. But at the same time, they make it excessively risky for the supplier in terms of working capital.

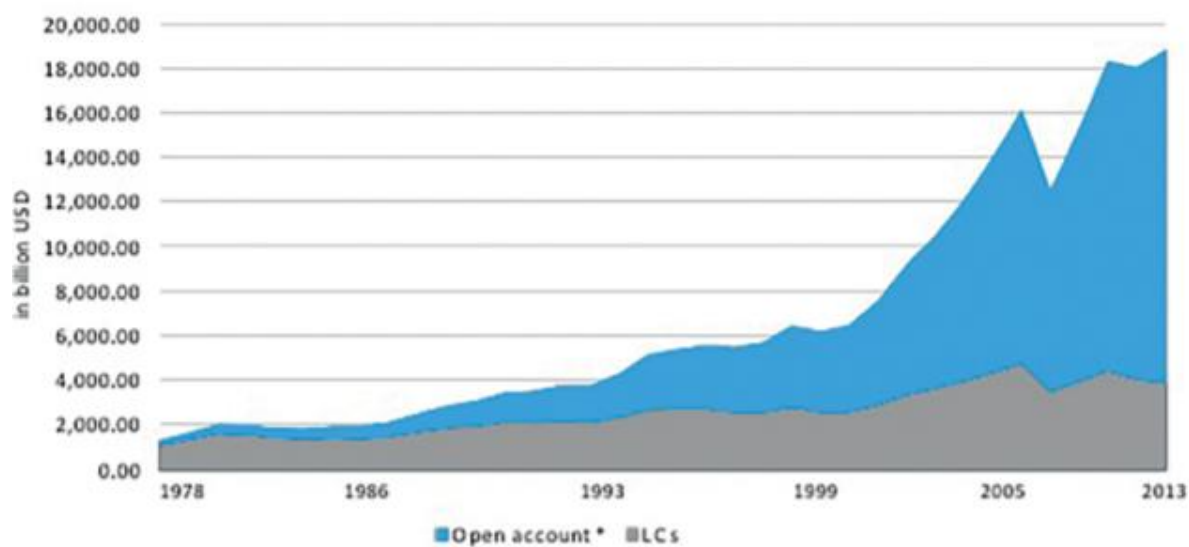


Figure 2.2 Growth in O/A vs L/C (Source: Global Supply Chain Finance Forum, 2015)

In this ‘safest’ form of import, the payment of goods is due, usually 1 to 3 months after the goods have been received by the importer. Due to high risk, suppliers do not prefer open account credit terms unless the importer or buyer has a long history of the business. However, to gain competitive advantage in the fiercely competitive market, many exporters are inclined to offer open account trade bearing an extra rise (EDC Canada, 2017). A clear trend in figure 2.2 shows the exponential increase in open account trade in recent years.

2.3 Types of Supply Chain Finance

There are different types of SCF models applicable to different types of businesses depending upon the business's model and its requirement. The business investigates the trigger events in its financial flow which can be used as collateral or security for the financing opportunities.

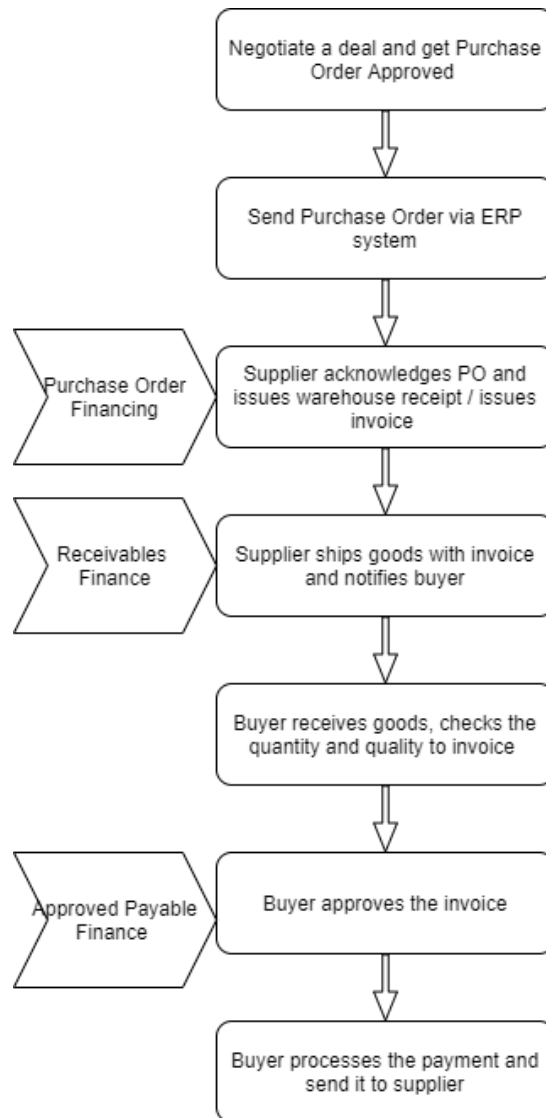


Figure 2.3 Trigger Events for Supply Chain Finance (adapted from: Lamoureux & Evans, 2011 and Bryant & Camerinelli, 2013)

Figure 2.3 demonstrates a typical financial process which begins with negotiating for a purchase order, followed by sending it to the supplier who acknowledges and starts manufacturing. Then it is shipped to the buyer with an invoice who checks the invoice and goods received with the purchase order and sends the payment to the supplier. The traditional process used for payment is very inefficient, as can be understood from figure 2.3 because it takes on average 55 days to complete.

Moreover, the transaction cost can go to several dollars as compared to a few cents when compared to automated invoicing (Berez & Sheth, 2007). These different trigger events lead to different forms of Supply Chain Finance models. As can be seen from figure 2.3, the financing can be either invoice based or purchase-order based, or post-shipment and pre-shipment. Invoice based finance can then be further classified into supplier-led or buyer-led.

2.3.1 Purchase Order based Finance

In this type of financing, as the name suggests, a purchase order is used by the supplier to finance its operations and labour costs to produce goods for a specific customer purchase order. It is mostly used by new companies to fund their operation that is in dire need of working capital or can be used by the supplier of a big company who suddenly is witnessing a big order like a government order. In these types of finances, the financing risk is higher for the financing party because of no 'physical' existence of collateral and involvement in the process at a very early stage. Purchase order finance can be classified into two subparts:

2.3.1.1 Pre-shipment Financing

Pre-shipment financing is the funding provided by a financing institution to a supplier to cover the supplier's working capital needs for producing finished or semi-finished goods and delivering it to the buyer. This is done by using either a purchase order, standby letter of credit or Bank Payment Obligation issued on behalf of the buyer. This type of financing involves the highest risk and disputes since the purchase orders are very likely to change and this adds complexity for financing provider and the seller. As a result, this is usually kept reserved for strong trading partners which have a history of consistent demand or orders. It can also be used for a sudden erratic demand given the agreement is followed tightly.

2.3.1.2 Inventory Financing

Pre-shipment finance is mostly observed in just-in-time businesses. Most businesses keep an inventory to increase their service level. In inventory finance, purchase order and inventory in stock or transit can be used as collateral for financing the additional working capital to restock inventory. The revenue generated can be used to pay back the loan and no personal asset is used for security. The financing is rarely done with a hundred per cent of inventory value, instead, the range is usually between 50 to 80 per cent for the liquidation value of the inventory. Advantages include funding for a seasonal spike in business and fluid lines of credit. However, there are some disadvantages as well. Due to high risk, the interest rate is higher over the repayment cycle when the inventory is finally liquidated when sold to a customer. Moreover, with time the value of the inventory most likely depreciates which leads to shorter payment terms that is equal to the lifespan of inventory. Inventory monitoring is a heavy task which leads the financiers to employ separate

personnel to monitor the inventory levels. This leads to extra costs for the financial institutions and delays in borrower obtaining the money. This has also led to double counting scandals, for example, the famous copper financing scandal in China which was a result of double-counting copper inventory (Chandrashekhar, 2014).

2.3.2 Invoice based Financing

According to a survey done on 20 million invoices by Fundbox, 64 per cent of small businesses have unpaid invoices (Eschenburg, 2015) because of open account trade terms and business seeking to extend their DPO. Using invoice-based financing, a beneficial solution can coincide for banks/lenders, buyer and seller. Suppliers can get rid of unpaid invoices by getting immediate payment by the banks/lenders. Bank/lenders have more security since it is involved in the later stage of the order and buyers can also experience lower risk because the production of goods has been acknowledged. Moreover, invoice-based financing can be easily used to incorporate the financing of services. As a result, invoice-based financing has acquired lion's share of financing in the past few years with 80-90% market share with remaining share to PO-based financing. Invoice financing charges a steady fee every month while PO-based finance increases the fees if customers don't pay on time and can take about one to two weeks to get financed as compared to two-three days in case of invoice-based finance. As mentioned earlier, invoice-based finance can be classified based upon the entity it was initiated by - buyer-led and supplier led.

2.3.2.1 Supplier-led Architecture

In supplier led invoice financing, the financing is done on the receivables of vendor/supplier. It can be done either via discounting on early payment of the invoice or factoring/selling the

receivables to a third party. In invoice discounting, business owners can leverage the value of their sales ledger. A certain portion of invoice becomes available to be financed by the lender, usually 80% to 90% depending upon the face value of the invoice, depending upon whether the buyer is domestic or foreign and if the receivables are insured. The discounting is also dependent on whether the money was lent “without or with recourse”. In the former case, the supplier is not sure whether the customer will be able to pay back the money and hence agrees for higher discounting and vice-versa. In case the buyer defaults on the payment, the lender cannot claim a residual amount from the supplier. Also, in the case of invoice discounting, the customer is not aware that the supplier has used tools to finance the cash flows and payment schedule stays normal. The supplier is still in control of sales ledger and can maintain the current standard of customer service. The efficiency of the operation is greatly dependent upon creating invoices and sending immediately to the bank to set up a fluid flow of operation which generates a consistent inflow of cash. The classical factoring of receivables works on the same principle with a major difference that the buyer is aware of the cash flow finance and the lender is in control of collecting money from the buyer. The lender might pay up to 75-80 per cent of the invoice to the borrower/supplier immediately after financing of invoices has been approved. After receiving full payment from the buyer, the lender repays the residuals amount after subtracting interest and charges. Like invoice discounting, factoring also comes in two forms: with and without recourse.

2.3.2.2 Buyer-led Architecture

In buyer-led supply chain finance the “anchor” is the buyer who initiates the process of financing. Like supplier led instruments, the financial institution provides the seller of goods or services a discounted early payment on the receivables but the difference being the finance is done based on

the buyer's credit risk. Buyer-led finance, also known as approved payable financing, is beneficial to the supplier when dealing with a bigger buying company with a bigger credit. The main reason that companies finance suppliers on their credit is to strengthen the resilience of the supply chain. This can be considered as an aftermath of the Global Financial Crisis which saw numerous supply chains collapse because of the supply shortages (Sheffi, 2015). By improving the health of the supplier's cash flow, bigger and costlier disruptions in the supply chain can be avoided.

In both the cases of invoice-based financing, receivables are purchased, and the title is transferred to the lender by a Receivables Purchase Agreement between the seller and the financial institution. But since the anchor is the buyer with bigger credit, the financial institutions face lower risk; consequently, the discounting rate is lower. By examining one buyer who has multiple suppliers, the process gets more efficient because just one credit check can lead to the financing of multiple supplying entities. Approved payables financing can be classified into 4 types:

2.3.2.2.1 Dynamic Discounting

This enables suppliers to obtain quick short-term financing directly from the supplier by offering buyers a direct invoice discount in case of expedited early payments. The main difference from receivables discounting from a third-party financier is that the formally rigid structure of discounting can be flexible with a variable discounting, depending upon the supplier's need and negotiation power. However, in this case, the buyer can face cash flow problems because the DPO has been shortened considerably.

2.3.2.2.2 Reverse Factoring

In this type of financing, once the invoices have been received by the ERP system of the buyer and approved as payables, the supplier is notified of approval via the financier. Following which supplier gets a discounted payment of the invoice by the financier at a better rate than dynamic discounting. This is because financing has been performed on the buyer's credit to pay back to financier later when the invoice has reached its maturity. There are two contracts involved with financial institutions: with the buyer and with supplier. The buyer will agree to pay "approved for payment" invoices and the seller will sign RPA or Receivables Purchase Agreement to transfer title of receivables. Figure 2.4 explains the process which begins with buyers initiating a payable program for all or portions of its payables to provide working capital to all or a section of the suppliers.

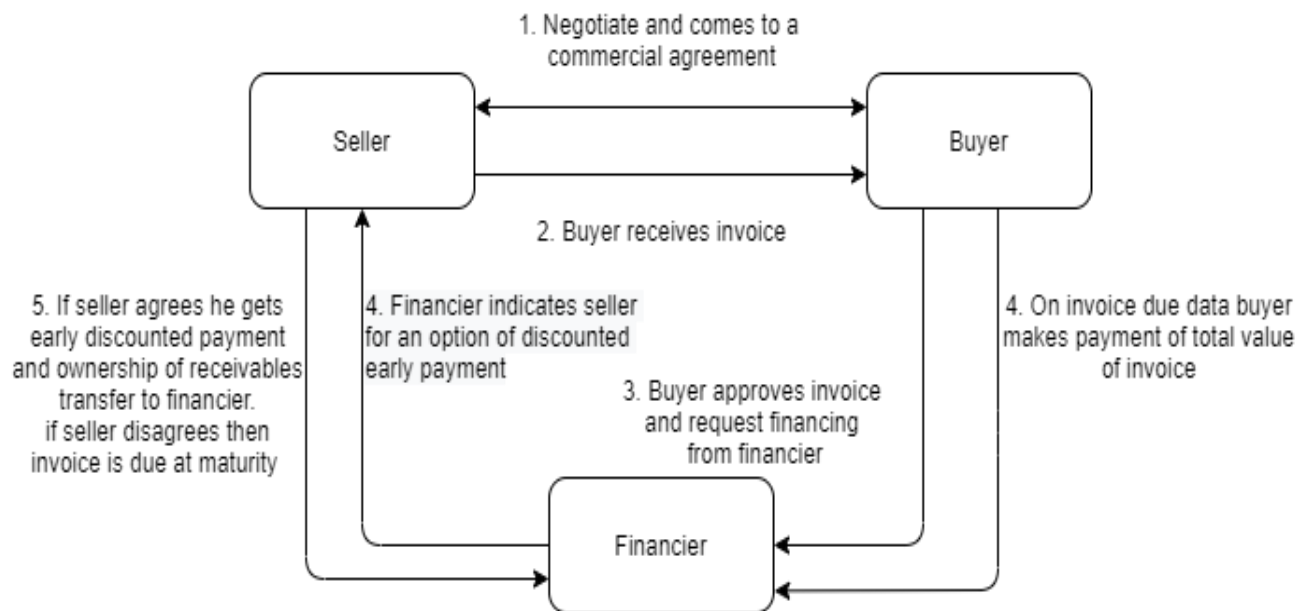


Figure 2.4 Reverse Factoring Procedure (constructed by author)

2.3.2.2.3 Reverse Securitization

Securitization process refers to exposing the capital to the market to reducing the risk by subsequently transferring it to the investors. In supply chain finance, supplier led securitization is the conventional way by securitizing the receivables by the supplier. The process begins by selling ill-liquid assets to the commercial bank/ issuer who gleans all the receivables bought from different suppliers, bundles them and sells them to an SPV or Special Purpose Vehicle in the form of a “true sale”. SPV converts assets into Asset Based Securities and exposes them to the market for investors. The receivables from various suppliers are bundled together by the SPV to form a diverse portfolio to reduce credit risk by diversifying the assets (Fabozzi et al., 2006). Suppliers get a lower financing rates because of competition in the market. Moreover, the cost of KYC reduces because now only SPV must be KYC’d instead of multiple banks/institutions doing it for every supplier which makes the process very inefficient.

In Buyer-led securitization or reverse securitization, instead of diversifying the investment by buying receivables from a pool of suppliers, all the receivables are from the same entity or debtor which approves payable in future. This makes all the credit risk concentrated to one enterprise which makes risk evaluation easier while simultaneously assisting suppliers having an early payment at buyer’s credit. In an aim to mitigate the concentration of the risk, the multi-buyer structure can be adopted where an SPV can be divided into multiple compartments or sections, each representing a different buyer. Figure 2.5 explains the process of reverse securitization.

After the supplier sends the invoice to the buyer, it can either wait for the buyer to pay the full invoice later or can have an early payment by reverse securitization. After the buyer has approved the payables and supplier requests early payment, the information gets relayed to payment agent.

Following this, a new commercial paper or note is sent to the central securities depository which pays supplier via SPV within 2-3 days at a discounted rate, determined by the rules of the government where it is operated. It is prerequisite for securities to settle before receivables are transferred to the SPV. After the buyer pays to the SPV after maturity of the invoice, the SPV provides principal and interest to investors.

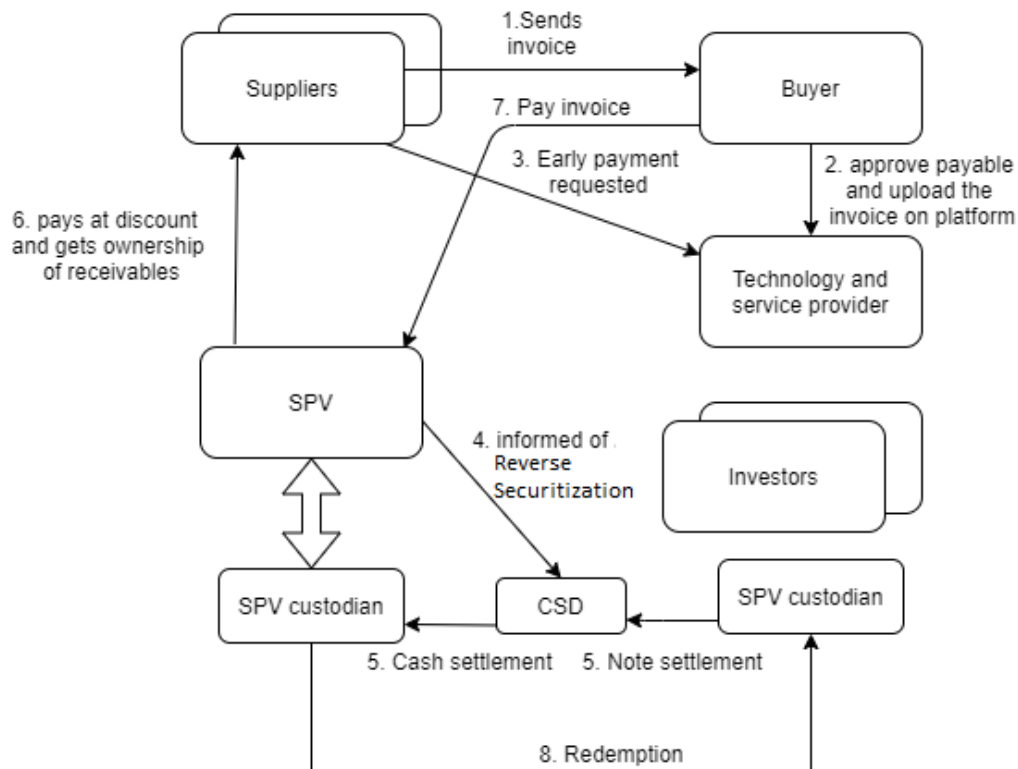


Figure 2.5 Reverse Securitization procedure (adapted from: Hofmann et al., 2017)

2.4 Key Drivers and Limitations of Supply Chain Finance

After the credit crisis in the 2008 financial crisis, Basel III framework was introduced for regulation, supervision and risk management of banks. At first, it was assumed that these new

reforms would prove harsh for SCF solutions. But the inherent liquidity of SCF solution posed them as a promising device to comply with new restricted capital and leverage ratios. In a global supply chain, on average there can be 40 to 80 documents interacting between dozens of companies (Noah, 2020). Complex systems like these can become costly and cumbersome for banks if they lack enough technological capabilities. Specialized fintech companies can speed up the transactions and make the whole supply chain more efficient. Another added benefit that comes with adding customers/suppliers is the strengthening of the network. With every new supplier joining the network there is a new lead which attracts a new spectrum of potential customers. For suppliers, it offers a possibility of achieving cash influx at an earlier stage than invoice maturity which can help them to sustain their operation and business. Reducing the DSO helps with better cash flow and if the finance is at the buyer's credit, the discount goes down further. For the buyer, who has the focal company of the supply chain with bigger credit, strengthening suppliers helps to make the supply chain less prone to disruption. Also, because of regulations, unpaid invoices are not treated as debt on the balance sheet which further helps in lowering of the financing rate. Moreover, if the buyer doesn't prefer transparency, it uses reverse securitization for early payments simultaneously controlling the purchaser of notes (Hofmann et al., 2017).

Even after all these benefits, when it comes to adoption of invoice based payable SCF instruments, they constitute only 20% because there are certain limitations which affect the adoption of SCF (Wuttke et al., 2019). There are three main barriers which hinder the adoption of SCF (Hofmann et al., 2017):

- a) **KYC:** A report by Euromoney indicates banks are terminating relationships with customers because of huge KYC costs on a single client - \$15000 to \$50000. Moreover, these costs

exponentially increase when the KYC is for a big company with multiple suppliers. KYC requirements are decided from a principle-based approach. Governments set up guidelines and regulations and banks determine by themselves how to comply with them. Hence, banks leave no scope to perform checks to prevent fraud and avoid billion-dollar fines (Financial Times, 2012). In an attempt to standardize KYC, attempts have been made to develop a central repository with all the checks of suppliers (Gustin, 2018). But banks continue scrutinizing firms since central depository always are vulnerable to attacks like hacking. A medium article (Wharf Street Strategies, 2019) identifies three main problems with current KYC processes: every bank conducts its KYC, in turn, increasing cost and time, every time a customer switches account a new KYC is done. The information is centralized and hence it can be altered.

- b) **Accounting treatment:** Accounting treatment is another problem because sometimes the buyer wants to divide the returns with the financier. This can lead to an increase in bank debts and hence limit the capability to obtain more credit. This can also have an adverse effect on the company's leverage ratio and loan covenants (PricewaterhouseCoopers, 2017). Moreover, trade settlement in the market can take 2 to 3 days after trading depending upon the jurisdiction because of numerous parties involved like custodians, CSD etc.
- c) **Transaction costs:** As it can be seen from the different SCF models, setting up a system like reverse securitization or reverse factoring requires multiple inherent transaction/steps to complete one transaction. This makes the effective transaction costs very high and the post-trade process becomes costly.

2.5 Where Does Blockchain Fit?

Blockchain, as defined by ISO 22739 (*ISO/FDIS 22739 Blockchain and Distributed Ledger Technologies — Vocabulary*, 2020) -

“Distributed ledger with confirmed blocks organized in an append-only, sequential chain using cryptographic links. Blockchains are designed to be tamper-resistant and to create final, definitive and immutable ledger records.”

Often used as a “buzzword”, it is touted as a panacea for every problem in different businesses. But blockchain holds a true potential in transforming the current structure of the financial system. As shown by a PWC study (Di Gregorio, 2017), blockchain can greatly cut down costs in financial services as it can be used in payments, capital markets, trade services, investment and wealth management and securities exchanges. It can help firms by reducing the cost of auditing and regulatory reporting. Post-trade reconciliation and settlement and many other processes which are currently manual can be improved by Blockchain. Using blockchain as a token of authentication will free up more space for employees to perform a value-added task, thus being beneficial to both employees and employer.

Different SCF instruments can be re-modelled using smart contracts to improve efficiencies by overcoming the adoption barriers explained in the previous section. Smart Contracts, explained in detail in Chapter 4, are short programs comprised of encoded business logic which run on a blockchain. But before delving deeper into how blockchain breaks down the barriers for SCM adoption, it is important to shed light on two characteristic problems that SCF solves in removing

credit market gap and how blockchain can aid SCF. Followed by this, how blockchain weakens the SCF adoption barriers and blockchain infused SCF models will be discussed.

Credit market gap denotes the phenomenon in which as the firm gets bigger, its ability to gain more credit increases and vice versa for smaller firms and thus making it a common phenomenon in exporting business involving various firms across the globe (Ragan, 2008). The problem becomes prominent in developing countries, for example, 40 million SMEs are denied of financing (Lekkakos & Serrano, 2016). The problem seems to be alarming when these SMEs contribute to 60% of total employment and 40% of their GDP (Stein et al., 2013). There are two causes of this credit market gap (Lamoureux & Evans, 2011):

- a) **Information asymmetry** – This refers to financial institutions having incomplete information about the suppliers of the focal company/buyer. Lack of information prevents them to finance because of the inability to assess the risk. Using SCF reverse factoring or reverse securitization, makes auditing easier and information more accessible since it mostly relies on buyer's credit. Blockchain's capability in solving information asymmetry problems is as famous as bitcoin itself. Improving the provenance by recording the transportation data on the immutable transparent ledger, creating self-sovereign identities for proving the authenticity of the supplier, and creating smart contracts based on trigger events discussed previously can dramatically reduce the information asymmetry problem. Not only reducing information asymmetry is beneficial for increasing supply chain resilience but it also makes the supply chain more managerially feasible (Devalkar & Krishnan, 2019). Using a blockchain-enabled reverse factoring arrangement can help to solve the moral hazard problem (entity increases its exposure to risk when insured for the

loss) while providing working capital needs for suppliers. Moreover, the propensity to adopt blockchain shoots up as the firm is facing more information asymmetry problem (Chod et al., 2019).

- b) **Positive externalities** – In a situation where the bank's cost of setting up a financing instrument between two or multiple parties exceeds its incentive (profit), banks will avoid financing. Financial institutions like any other company justify their existence with corporate profits rather than social benefit. There exists no common platform to a huge spectrum of SME because of higher administrative and set-up costs (Lamoureux & Evans, 2011). In sectors, like pharmaceuticals, where administration costs are high due to regulations on the product, the problems are worse. In a joint report by Maersk and IBM, administration and processing costs were estimated to be as high as 20% of the transportation costs (Mark B. Solomon, 2018). In the finance domain, these costs are mainly in KYC and warehouse auditing. Automating the processes using a trustable decentralized ledger can greatly reduce the problem (*Blockchain Use Cases For Banks In 2020*, 2019).

2.5.1 Overcoming SCF barriers

- a) **KYC**: It has been already established that KYC is one of the major problems for financial institutions. Using blockchain, a single immutable source of the decentralized ledger can be developed which allows removal of redundant checks. Data can be stored on-chain or off-chain and blockchain-based DApp (an application made on Blockchain) can be used to access the data any number of times. Moreover, it renders the data an exceptional capability to be tamper resistant and entities can have control over their data which can only be shared

after their recorded consent. Instead of doing KYC multiple times, the company's first KYC can be used to upload its details on the blockchain-platform and then a unique ID can be used to access the details every time KYC is required. The current cost of KYC is very high, so buyers and financiers have the propensity to only bring important suppliers onboard. Using blockchain, KYC will be made cheaper and hence SCF can be used at its true potential. However, it would be ambitious to think KYC would be completely transformed after integrating with blockchain. A Goldman Sachs report stated that the banks would continue extensive KYC checks since they are still liable (Goldman Sachs, 2016). Blockchain would return no benefits if the KYC's are done by a single authority. An extensive network of multiple banks and validators will be required so that a shared ledger can be made.

- b) **Accounting Settlement:** Originally meant to be a distributed digital accounting set-up, blockchain can avoid the repeated cycle of segregating and merging the records in different compartmentalized databases by using smart contracts on a single immutable one. It can transform the way auditing is done. Accounting records will be cryptographically available secured and discoverable on the ledger. To check integrity while maintaining anonymity will be possible now by using the hash generated by the electronic/virtual document. If the hash matches, the ledger/accounting is untampered. This process will be considerably stronger than the current process.
- c) **Transaction Costs:** Blockchain's capability to reduce transaction costs is well known. Only transaction costs are the one that is paid to miners to validate transactions. In Ethereum, the transaction fee is called "gas", explained in detail in Chapter 3. Since there is no central authority who has the job to manage records, update and secure them, the

transaction costs are extremely less as compared to the current system. But it must be noted that this comes at “cost” of reduced transaction speed. But an incredible amount of research has already been carried out to scale blockchain which will be discussed later in Chapter 6.

Chapter 3: Blockchain Technology

3.1 Introduction

The term “blockchain” has been ubiquitous all over the internet after the advent of Bitcoin. A Google Trend on the search terms “blockchain” and “Bitcoin” shows the interest over time peaked in Dec 2017 (*Google Trends*, 2019.), during the same time when Bitcoin reached its highest price approximately \$19000 (*Bitcoin Price Index — Real-Time Bitcoin Price Charts*, 2020.). But the same chart also shows that the interest suffered a setback when the bitcoin price “bubble” was burst. Figure 3.1 shows the viewer that after the price drop all the research and development in the blockchain space might have come to a halt but that is not the case. Interestingly, there had been numerous reports and articles all over the internet about the scale of disruption blockchain is expected to bring to the business world. Numerous articles have been published by prestigious publications like Forbes - it is believed that the capability of blockchain to make secure digital partnership will make it extremely crucial to a business (Marr, 2019 & Olenski, 2018). Only because organizations are struggling currently to scale the distributed systems globally, it appears that there is not enough movement in the blockchain world (Marr, 2019a). But a lot of innovation is going on to make the blockchain more secure and easily scalable. For example, IBM and Maersk have already been successful in developing TradeLens to streamline global shipping operations across the globe and significantly reduce paperwork. Inferring from a CBN article (CBN, 2020), it is believed by various crypto-investors that bitcoin will enter a bull run and hence we can expect the same trend towards blockchain investments. And the current investment in cryptocurrency during the COVID-19 pandemic in an attempt to find an investment safe-haven attests to this idea (Josie Cox, 2020).

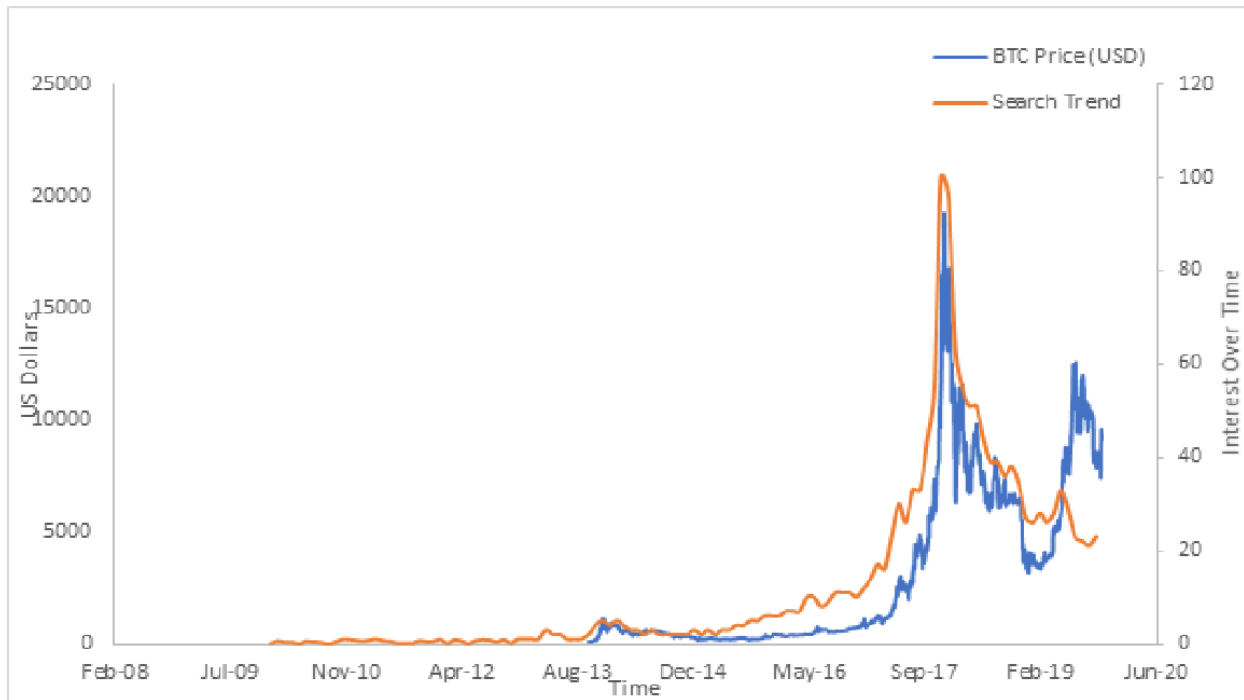


Figure 3.1 Bitcoin price vs Blockchain Google Search

To predict the future of anything, it is important to investigate the past. And the history of blockchain can be considered synonymous to the history of bitcoin. Bitcoin.org was registered on 18th August 2008, followed by the open publication of the famous ‘white paper’ by Satoshi Nakamoto about its construction. Bitcoin worked on a sophisticated technology involving the use of public distributed ledger, timestamping, cryptographic hashing and proof of computational work to reach a consensus. This was used to make immutable records of transactions or blocks on a distributed ledger which coined the term ‘blockchain’. Figure 3.2 illustrates the chronology of inventions and publications in blockchain and crypto space. Beginning in late 70s with the development of concepts of asymmetric cryptography or hashing, followed by byzantine fault tolerance, proof of work mechanisms and timestamps lead to the development of today’s very secure blockchain. But if we inspect the figure closely, one of the most important papers on smart

contracts by Nick Szabo (1997) led to the development of a new kind of blockchain Ethereum by V. Buterin (2013).

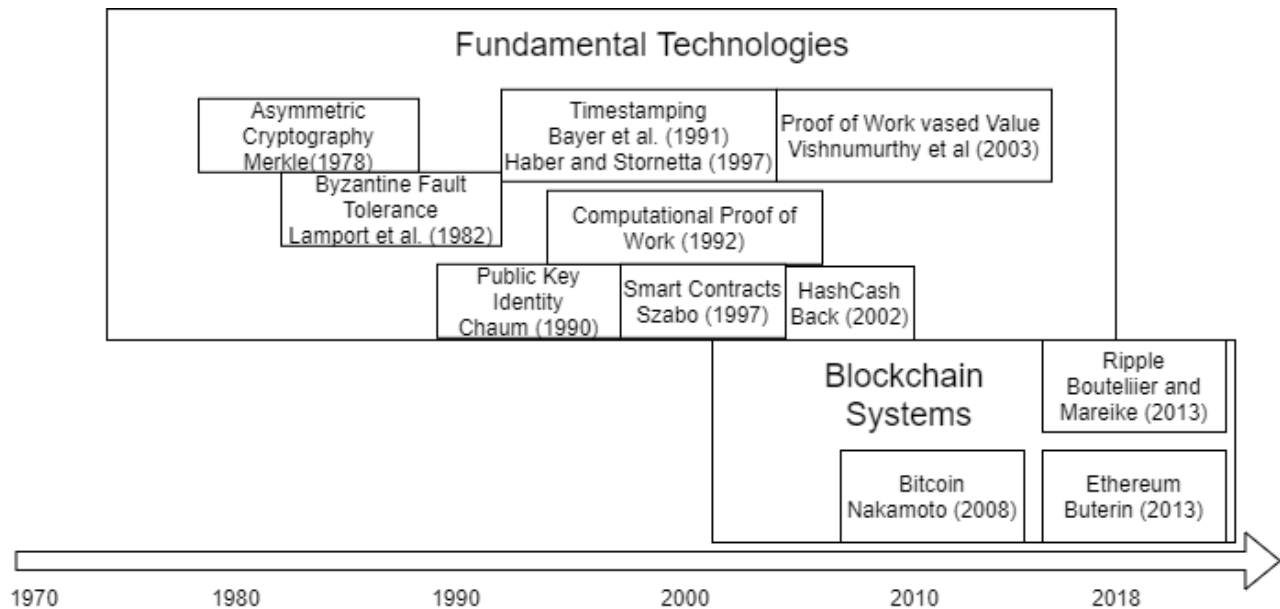


Figure 3.2 Timeline to Current Day Blockchain (adapted from: Kaligotla and Macal (2018))

3.2 Types of Blockchains

To delve deeper into the blockchain, it is crucial to know the distinction between the different types of blockchains. According to ISO 22739 (ISO/FDIS 22739 *Blockchain and Distributed Ledger Technologies — Vocabulary*, 2020), blockchains can be classified into four quadrants as shown in figure 3.3. The four quadrants are generated by

- Access: permissioned - requires authorization to perform activities, or permissionless – not requiring any kind of authorization for activities
- Validation: public- accessible to the public for use, or private – accessible to only a limited group of users.

The first quadrant engulfs all the distributed ledger systems (or blockchains) that are accessible to any user. Any entity anyone having internet connectivity can become a node on the blockchain and can have access to past and current records, mining blocks, verifying transactions etc with no authorization required. Some examples are Bitcoin, Litecoin, Ethereum etc.

The second quadrant encompasses all the distributed ledger systems that are accessible to the public, but the activities can only be performed after authorization. For example, Hyperledger Indy platform is a decentralized identity system which can be used to generate self-sovereign credentials (credentials with the self-control of data). Developers can make applications on top of it for public use but the activities on those applications are not “permissionless”. For instance, a user can only generate a credential but cannot issue it. It can only be issued by a specific trusted entity whose integrity can be checked by everyone using the entity’s DID or Decentralized Identifier.

The third quadrant encompasses all the distributed ledger systems that are privately governed but can be used by any user. Instead of having a mining algorithm, the inner working of these blockchains could be a bit different. An example is Ripple (XRP) which a blockchain platform to exchange Ripple or XRP but it is governed centrally by the Ripple Organization and any changes on it are brought solely by the decisions from this organization.

Finally, the fourth quadrant encompasses all the distributed ledger systems that require authorization by to onboard users. The user activities and access are also controlled by smart contracts, explained in detail in chapter 4. For example, Hyperledger Fabric provides businesses with tools to make a private blockchain for their supply chain network. Only after the permission, the entities can be onboarded on the network. Despite using a distributed ledger system, all the

transactions are secure but not transparent. Confidentiality of transactions can be maintained within the network entities using the channels, and outside the network using the authorization access.

	Permissionless	Permissioned
Public	Bitcoin Ethereum	Hyperledger Indy Sovrin
Private	Enterprise Ethereum Alliance Azhure Ripple	Hyperledger Fabric Hyperledger Sawtooth R3 Corda

Figure 3.3 Types of Blockchain

3.3 Components of Blockchain

There are five most important characteristics of a blockchain discussed as follows.

- a) **Distributed Ledger:** Since ancient times, humans have been involved in exchanging assets, transacting food and precious metals. With the advent of a banking system, banks became a central authority to manage these transactions and store the money. Followed by the internet boom banks began to manage and control all the online transactions. The idea

of distributed and decentralized ledger served two purposes: a more robust system for cyber-attacks and original intention of the decentralized web, maintaining transparency with security (Kadam, 2018). In other words, distributed ledger contains the transactions packed in a chronologically ordered linked list like structures, also called as blocks, making a blockchain. The system reaches to a peer-agreed state by a consensus mechanism among all the participants operating as nodes on network, which can hold a local copy of the ledger on their machines.

- b) **Hashing:** Hashing is the first level of security in the blockchain to preserve the immutability in the chain. In hashing, mathematical operations are used to convert an input of any length into an output of definite length. But in addition to this, there are several properties required by a Hash function to be useful in blockchains. It should be:
- i. Deterministic- It should produce the same output every time for the same input.
 - ii. Fast in performing computations – It should be fast considering the several numbers of times it will be applied to construct a block and digitally sign every transaction.
 - iii. Unidirectional – The output of the hash cannot be used to determine the input.
 - iv. Collision free – It is computationally infeasible to find for a given input a second input that maps to the same output.
 - v. Sensitive – A useful hashing function should be very sensitive to the input. A small change in the input must cause a total change in output.
 - vi. Puzzle friendly- For mining purposes, a hashing algorithm should be “handy” to make puzzles, like those used in Bitcoin to find a number which produces a hash value containing a specific number of zeroes.

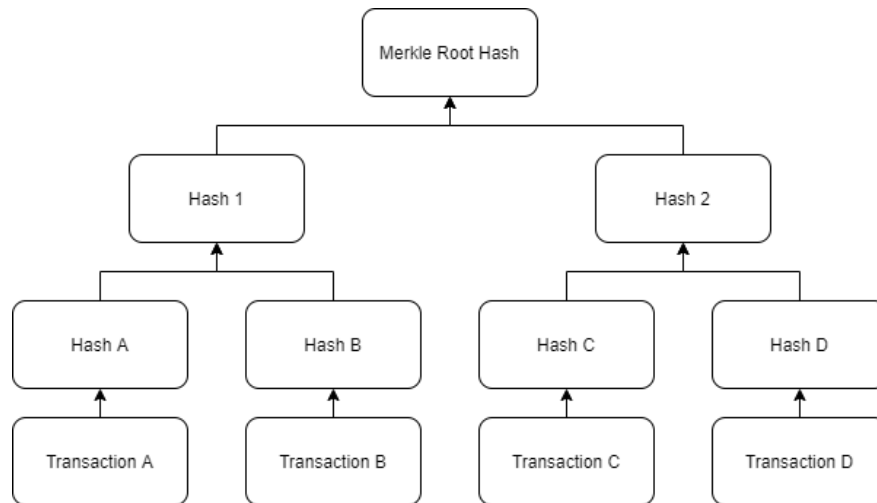


Figure 3.4 Merkle Tree Structure (constructed by author)

Different hashing algorithms are used in different blockchains to construct Merkle trees. For example, bitcoin uses SHA-256 while Ethereum uses Keccak -256. Absence of the same hashing function is one of the biggest barriers to the interoperability of different blockchains. Merkle tree is made by hashing of all the nodes in pair (here a node is a hash of transactions in a block) until there is only one hash left, also known as Merkle root or Root Hash. It represents a cumulative hash of all the transactions in the block header. Merkle root hash allows a quick check of the consistency of all the transactions instead of checking them individually. Figure 3.4 shows the hashing structure in the Merkle Trees.

- c) **Asymmetric cryptography:** To understand the concept of asymmetric cryptography, it is important to understand symmetric cryptography first. Symmetric cryptography uses sophisticated mathematical equations to encrypt the data by a single key. The sender uses the key for encryption and then the receiver uses the same key for decryption, for example, a password protected pdf sent over the email. But in asymmetric cryptography, a key pair of the public and the private keys is used for advanced security. The private key is never

shared in blockchain and used to produce a public key using a specific algorithm. Several algorithms can be used for this purpose but usually, all blockchains employ Elliptical Curve Digital Signature Algorithm or ECDSA. The main purpose of using the public key is to authenticate the transaction to the appropriate sender without revealing the private key.

The process begins with hashing a message or a transaction that is then sent to the receiver's public address, which is a hash of the receiver's public key and some additional data. It is then encrypted by the sender's private key. This encrypted information together with the hashing algorithm is a 'digital signature', which proves the origination of a transaction and "locks" its content to assure the integrity of the message. Once the message or transaction is received by the receiver, it decrypts it using the sender's public key to obtain the hash. Simultaneously, the receiver calculates the hash of the message/transaction and if both matches, the authentication is successful.

- d) Timestamping:** Timestamping's function, if judged from the name, gives a false idea about its true purpose in developing a secure blockchain. To explain this, it is crucial to understand how data or transactions are stored on the blockchain. Although this will be explained elaborately later in Section 3.4, to have an initial high-level understanding is beneficial.

Transactions are grouped together to form a block and hashed by the miners to form a Merkle tree. Miners compete to solve a computationally intense problem to "mine" the block by finding the hash value of the block lower than a certain "difficulty" (number of zeroes required in the total hash result of block-header) that is decided by the network.

The moment when the problem is solved, the block creation is timestamped. This property is prone to attacks because the accurate time of the events or the "strong freshness" of times

on blockchain is “questionable” (Szalachowski, 2018). This is because timestamps can vary in hours from the timestamps of nodes and can be dramatically different, in theory, from the actual time. This restricts blockchain’s ability to be a time record. Order, validity and integrity of blockchain can be achieved by its property of hashing and linking previous block header's hashed to the current block header’s hash.

Timestamping serves two crucial functions – difficulty calculation and locktime transactions (Yannik, 2019). For difficulty calculations, timestamps in previous blocks help to determine the actual time taken to mine them. This, in turn, helps the nodes to determine the right difficulty to be used for subsequent blocks so as to maintain a 10 minutes inter-block time. Since miners have the leverage and incentive to manipulate the timestamp, it will not change the difficulty since the time will be checked against real-time. The following conditions must be met for a timestamp to be valid: it should be within 2 hours of network adjusted time (median of all timestamps returned by all peer nodes) and it should be greater than the median of past 11 blocks.

Locktime refers to blocking a transaction to mined before a certain period which can be done using either blockchain’s height (total blocks in the chain) or unlocking it at a specific time using the timestamp.

- e) **Consensus and mining algorithms:** The idea of blockchain, cryptocurrency or any decentralized system came into existence only because of the major issue with “trust”. In the case of a centralized system, the trust is levied upon the central authority but in a decentralized and distributed system, such single bearer of the trust is absent. The solution to the problem is maintaining a single record of history to which all the nodes agree. The true need for consensus is to evaluate and agree about all the data and addenda before

everything becomes permanently stored on the blockchain (Mingxiao et al., 2017). To accomplish these objectives, blockchains employ different consensus algorithms. Following are the most popular consensus algorithms:

PoW or Proof-of-Work: This algorithm is used by Bitcoin to facilitate transactions and block creation every 10 minutes. As described above, hashing algorithms play a crucial role in securing data or transactions. Once the transactions are bundled for a block and Merkle Hash is calculated, the miner performs computations to find a ‘nonce’, an abbreviation for a 32-bit “number used only once”, to find the block header’s hash. The nonce is stored in the block-header along with important data attributes like the Merkle Hash. The aim is to find a 256-bit hash containing a specific number of zeroes - depending upon the preset difficulty level. If the hash does not have enough number of zeroes, then the hash is discarded, and operation is begun again until the hash with the required zeroes is found. This activity of finding the nonce makes blockchain extremely secure. The amount of computation power required to insert fake transactions increases exponentially with the length of the chain, sometimes even more than the incentive of introducing a fake transaction. Later, since the system is decentralized, a copy of the original ledger is kept on every node. In case a fake ledger competes, its authenticity can be quickly dismissed.

PoS or Proof-of-Stake: PoS was generated as an alternative to PoW due to the need of having a low cost, low energy consuming algorithm. To be adopted by the Ethereum 2.0, creator of the block is chosen randomly or through coin-based selection. It combines the lowest hash value of the chain with the stake or the amount invested in the block being mined. Validators instead of miners are used in this algorithm since there is no need to solve complex computational problems. The idea behind using PoS was originated to solve

the problem with the formation of Mining pools to mine bitcoins. Once the total computing power of a mining pool goes above 50%, the attackers would be now able to manipulate transactions and perform double-spending, also famously known as “the 51% attack”. Since, validators are chosen based on the total wealth in the asset being validated and receive transactions fees after validation, higher the wealth at stake, the greater probability of being chosen for validating the specific set of transactions and consequently mitigating the possibility of 51% attack.

PBFT or Practical Byzantine Fault Tolerance: A problem of distributed computing is the Byzantine fault, which results in information on different nodes being out of sync. A component of the distributed computer system is at the Byzantine fault if it fails or has imperfect information on it. For example, a server can appear to be perfectly working to some nodes of the asynchronous system while it may be caught by a failure-detection system and some nodes may regard it as a failed server. Such a problem arises because of the system’s incapability to reach a consensus about the server’s state. Drawn from the famous Byzantine general problem (Lamport et al., 1982), Byzantine fault tolerance is the system’s capability to deal with such problems. In this algorithm, a general or a leader receives a message from a client to invoke a service operation. The leader node uses the message along with the state to perform computation or the operation. A consensus is formed after the general or the leader node shares the decision with other nodes as well. Although no asynchronous system can always guarantee complete consensus as shown by famous FLP Impossibility (Fischer & Lynch, 1985), BFT assures a practical solution to cases like Byzantine faults.

3.4 How Does Blockchain Work?

Every component in blockchain has a very specific task to perform whether it be time stamping or hashing, but the goal is to have a strong and robust decentralized distributed ledger system. Suppose Peer A invokes a transaction to Peer B. Now, unlike a traditional centralized database system, the transaction must be transparent and secure. To achieve this blockchain offers a characteristic property to permanently append the transaction on an immutable ledger. Figure 3.5 explains the working of blockchain. Peer A begins by signing into his/her wallet using his private key or mnemonic(used in metamask, refer: <https://metamask.io/>). The peer then finds the public key of Peer B and sends the transaction. Similar transactions bundle up for T minutes (specific to the blockchain) and then they are broadcasted to miners.

Miners bundle them up further randomly or according to gas price, a transaction fee that a user decides to pay to the miner in Ethereum for validating the transactions. Once the transactions are bundled in a block, their hashes and a subsequent Merkle Root hash is calculated to append to the block header along with the hash of the previous block's header to create the typical, linked list-like structure. This is followed by the miner's actual job to solve the puzzle for the block or in simpler words, find the nonce which satisfies the current requirement of producing a hash with the specific number of zeroes. The number of zeroes is determined according to the difficulty set by the network using timestamps. After the mining completes the miners receive a reward depending on the blockchain, 12.5 BTC or 3 ETH and broadcast the blocks to rest of the nodes.

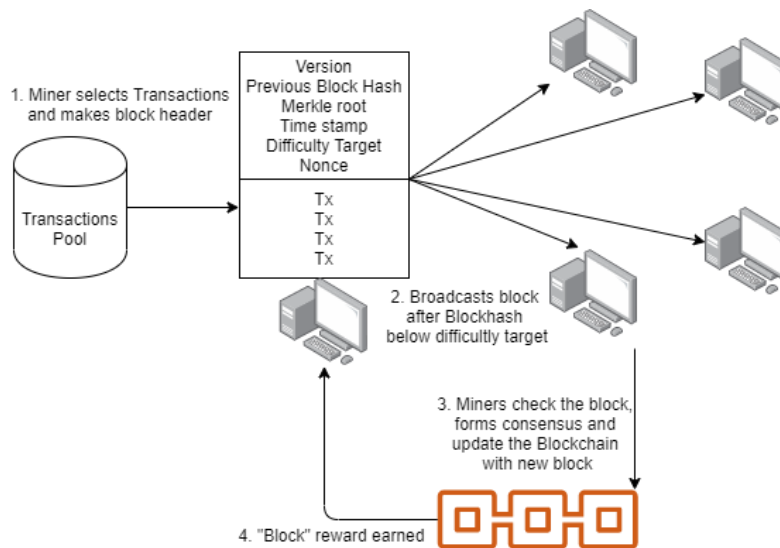


Figure 3.5 Working of Blockchain (constructed by author)

Block receiving miners use the nonce to check the hash to find if tampering was done with transactions, and if passed the block is finally added to the blockchain at every local node. It must be noted that the rule to work on the longest chain is maintained, in case two blocks are mined at the same time. All the miners validate the integrity of blockchain, thus forming a consensus.

3.5 Advantages and Limitations of Blockchain

Blockchain is advantageous to solve the issues that arise due to lack of trust. But at the same time, it would be reckless to use blockchain in almost any setting while ignoring the disadvantages it has. Gatteschi (2018) very elaborately wrote about the advantages and disadvantages.

3.5.1 Advantages

a) Immutability and robustness: Blockchain provide a shared ledger containing transactions that have been onerously reached to a consensus after complex computation using hash functions and

nonce. If anyone with a malicious intent tried to tamper with the information, given the current state of computation power available, it is practically impossible to change and delete information. It is permanently recorded on the blockchain.

b) Trustless: Use of digital signature moves the requirement of the trust from people to the technology. Use of private and public key makes it a sturdy source of anonymous trusts.

c) Decentralization: In public blockchains, there is no central authority which monitors and approves the transaction. There is no single point of failure where the power is concentrated. So, no single entity has the power to shut it down or change it.

3.5.2 Disadvantages

a) Energy consumption: Consensus algorithms like PoW are required to do an extremely high amount of computation to reach consensus. These computations are done on specialized machines owned by miners which compete to solve the computational problem. Since there is only one winner, a major amount of resource is wasted, which is electricity along with the opportunity cost of using the hardware. To comprehend the scale of the problem, estimation could be made based on the fact that bitcoin mining uses more electricity than the entire country of Switzerland (Vincent, 2019).

b) Transactions speed: Having a central authority to approve transactions like VISA can have a significantly higher number of transactions per second than a decentralized system. This is because reaching consensus requires a significantly higher amount of time.

c) **Fault in Smart Contracts:** Smart contracts are a short computer programs which can automate operations on blockchains (explained in detail Chapter 4.) These smart contracts need an API (Application Programming Interface) or an oracle service to upload data, which should be reliable, on these blockchains. This creates a trust problem because if the data coming from the Oracle is not reliable, then the smart contract will not function properly. Moreover, smart contracts can be buggy, and this could be exploited by malicious hackers as seen with the infamous DAO hack in June 2016, where a malicious actor was able to use recursive call attack to syphon out Ethers worth 50 million dollars at that time (Dhillon et al., 2017).

d) **Immutability and transparency:** Although immutability and transparency are touted as one of the key functionalities in blockchain. This can, at the same time ,go against the fundamental right of privacy and the right of erasure (“Art. 17 GDPR – Right to Erasure (‘Right to Be Forgotten’),” 2016).

As a result, it is important to look at both sides before adopting blockchain for use. Some question must be asked before adoption as mentioned by (Gatteschi et al., 2018): Is there even a need for decentralization or Public ledger or removal of disintermediation or independence of trust on people?

Chapter 4: Smart Contracts

4.1 History

As a result of dissent towards current structure of centralized and surveilling governance, several crypto-anarchist communities were formed in late 90s and early 2000s, like Cypherpunk, which believed in a system of self-regulations using anonymous, peer-to-peer and decentralized instruments (Chohan, 2017). Bitcoin proved its capabilities to the crypto-anarchist communities, by pseudonymous, trustless and peer-to-peer transfer of money eradicating the need of bank as the intermediary (GMT, 2012 and Petersson, 2018). At the same time, various developers and professionals began testing its underlying technology in numerous other domains like securities exchange, healthcare data exchange etc. But it required use of complex codes which can interact with blockchain and hence smart contracts came into being.

Blockchain makes smart contracts more robust and trustworthy, at the same time smart contracts makes the blockchain more functional and versatile – a perfect symbiotic relationship for a trustless ecosystem. As we have already seen from the timeline from figure 3.2, the concept of smart contract dates back to 1997 when Nick Szabo, a computer scientist, cryptographer and lawyer developed them intending to implement the contractual agreement of law and to facilitate buying and selling of goods between strangers using a system similar to bitcoin known as “bitgold”, believed by many to be the precursor to it (Szabo, 1997 and Szabo, 2008).

Bitcoin, although a Turing-incomplete language, a programming language that does not support reading and writing memory and infinite looping capability, could be used to develop smart contracts on top of it (BTC STUDIOS, 2017). However, the process is very cumbersome because

one needs to possess the knowledge of opcode programming. This limitation created a need to a whole new blockchain system known as Ethereum by Vitalik Buterin (*A Short History of #Ethereum*, 2019). In Ethereum, smart contracts escaped the limited scope of a legal contract and they transformed into an open-source and secured procedure which can be used to securely transfer value via tamper-proof means. To make the process of creating smart contract easier, Ethereum not only developed Solidity, a programming language of its own but it also adopted changes in its very basic structure, for example, to use more intuitive account-balance based transaction approach rather than UTXO's (Unspent Transaction Output) based as employed in Bitcoin where all the unspent bitcoins determines the current balance (Buterin, 2015). Moreover, Bitcoin's scripting language doesn't support complex logic because it only allows basic arithmetic, logical and cryptographic operations.

4.2 Definition and Operations

To provide a technical definition of smart contracts and where they fit in the blockchain, it is important to look at the simplified blockchain protocol stack as shown in figure 4.1. The blockchain layer sits on the top of the TCP/IP internet layer as the foundation and below the application layer (Rosa et al., 2019).

- a) **Peer-to-peer protocol** : In the peer to peer protocol, blockchain might randomly select a peer, for example Bitcoin, or can use a different protocol Kademlia like UDP (User Datagram Protocol) to select a peer, for example Ethereum, and spread the information using a gossip mechanism so as to ensure information disseminates to all the peers or nodes (Gencer et al., 2018).

- b) **Consensus:** As already discussed in Chapter 3 of Blockchain, consensus algorithm has the job to bring all the nodes of the blockchain on the same final blockchain state.
- c) **Transaction layer:** The transaction layer sits on the top of the consensus mechanism which records transactions and bundles them up in the blocks.

Smart contracts are pieces of codes that interact with the transaction layer. They contains a transparent procedure that is automatically verified, executed and enforced when certain conditions are met.

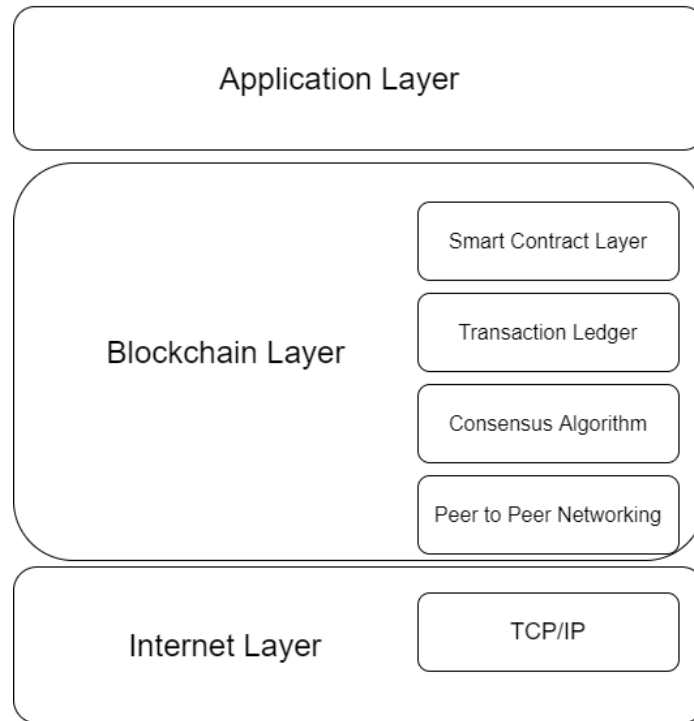


Figure 4.1 Blockchain Protocol Stack (adapted from: Rosa et al., 2019)

Users can interact with smart contracts by their externally owned accounts or EOA in a distributed application or DApp, for example, Idex or Cryptokitties (Floyd, 2018), whose smart contract has

its own contract address which can store ethers. This contract address is often attributed as the contract account. Once the EOA triggers a transaction, if the recipient is a zero address “0x0” it leads to the generation of the contract. After the generation of the contract, the smart contract address is used to run its associated code which is described further.

Smart Contracts utilize two things to perform operations: state and value. It is comprised of various functions to invoke operations/transactions using an “if-else-then” architecture by triggered action from the user. Once the transaction is invoked, it is disseminated via the blockchain-specific p2p network protocol, for example, the UDP-based discovery mechanism derived from Kademlia used by Ethereum (Maymounkov & Mazières, 2002). The transaction contains all the appropriate details, for example, contract address on which it has been made, and the concerning parties between whom it occurred. It is then verified by the miners for a specific fee and stored in the blocks. In the case of Ethereum, the fee is denoted by the term “gas” (Wood, 2015). Once the miners are incentivized by the “gas” fee, they execute the contract “ABI code” or in simpler language, create the contract in a blockchain specific execution environment. Ethereum was the first permission-less blockchain with a full Turing-complete execution environment known as EVM or Ethereum Virtual Machine. Each miner hosts an EVM which compiles the Solidity code to an ABI-code or byte code for execution. Once a trigger action is provided with some value and if the conditions in contract logic are met, the contract changes the “state” of the system with transaction. Once the transaction is validated, it is put into the blockchain as soon as the consensus mechanism approves it. Figure 4.2 shows the operations of a smart contract in a schematic manner.

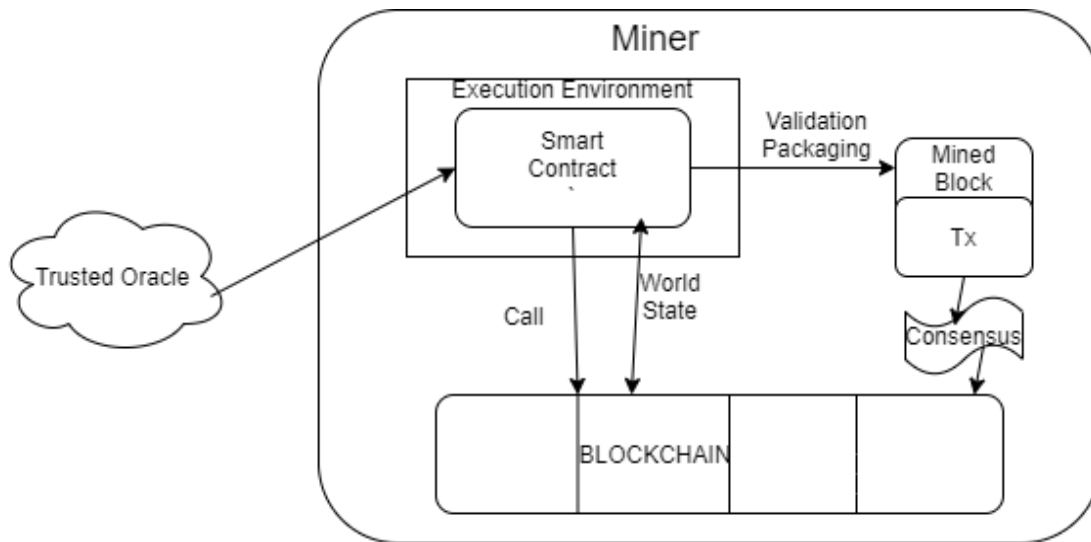


Figure 4.2 Working of a smart contract (adapted from: Wang et al., 2019)

Already discussed as a limitation of Bitcoin, it must use side chains to deploy complex contracts which can support supply chain tracking or supply chain finance making it impractical for make smart contracts for complex systems. But it is important to delve deeper into private and consortium blockchains like Hyperledger and their working so that an informed choice can be made for developing the proof-of-concept.

Hyperledger project, started by the Linux Foundation is the most famous private blockchain. Backed by several famous companies like IBM etc. the Hyperledger Project encompasses various sub-projects like Hyperledger Fabric, Hyperledger Indy etc. which together makes the Hyperledger Greenhouse system. The peculiar capability of Hyperledger is the modularity because different projects can be combined for use. For example, Hyperledger Indy and Aries can be combined to make identity management based permissioned-blockchains. For supply chain practices, the focus will be on Hyperledger Fabric due to its exceptional capability to scale and perform well with supply chain operations. Hyperledger Fabric allows specific organizations to join the network by

membership service providers. These organizations eventually become peer nodes which contain a ledger and contracts, also known as chain codes. In a similar manner to Ethereum which uses EVM, Hyperledger uses special containerized environments called Docker to run the chain codes (*Hyperledger_DataSheet_11.18_Digital.Pdf*, n.d.).

Hyperledger Fabric's operational procedure can be divided into three parts (Wang et al., 2019):

- a) **Proposal:** The user uses an application to invoke a part of the smart contract, also called as chaincodes. This is sent in the form of transaction proposal to all the organizations as endorsers to which endorsers respond with information on value, read set and a write set, list of unique keys and version number that the transaction wants to read and write, along with their cryptographic signatures. It must be noted that chaincode runs only in this phase. One difference to point out is that Hyperledger does not have any concept of mining or "gas" to prefer one transaction over others.
- b) **Packaging-** In this part, the transaction is verified by checking the signatures of endorsers and the coherency of the proposal responses. Once verified, they are bundled up in the blocks and propagated to all the peers via orderer.
- c) **Validation** - After the peers receive the blocks, all the peers reverify every transaction to check if it has been correctly endorsed by the respective organization that was decided in the protocol.

4.3 Research Models for Smart Contracts

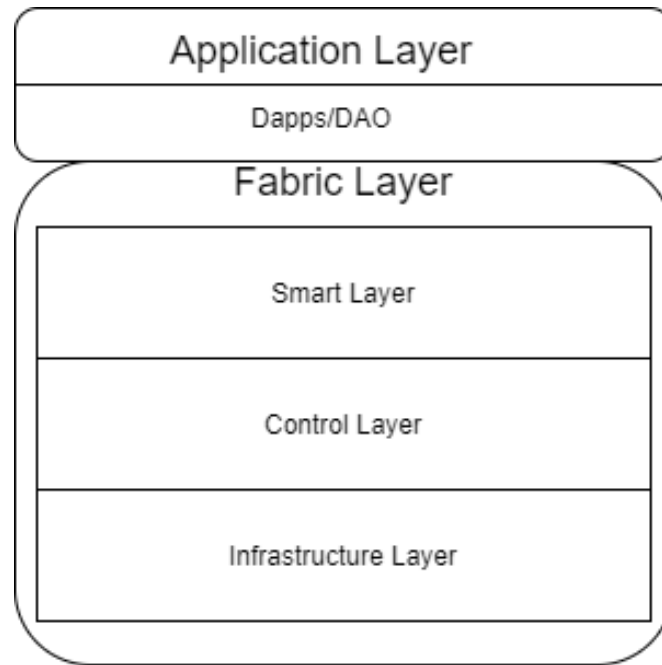


Figure 4.3 Blockchain Research Model (adapted from: Wang et al., 2019)

In this section, we will discuss various research frameworks that have been developed by researchers that can be used to evaluate the applicability of smart contracts and further pave the way to estimate their limitations and risks. Wang (2019) stacked six layers on top of each other to provide a blockchain-based solution. It consisted of an infrastructure layer, contracts layer, operations layer, intelligence layer, manifestation layer and an application layer. On the other hand, other some researchers bundled up the last five layers into a “fabric layer” working in conjunction with an application layer (Glaser, 2017). But to have a comprehensive outlook at the smart contracts, figure 4.3 shows a hybrid description of their model to assist in choosing a blockchain and developing the required solution for supply chain finance proof-of-concept which is described

in the next Chapter. For this hybrid approach, the fabric's layer will contain infrastructure, control and smart layers.

- a) **Fabric layer:** The fabric layer makes up the blockchain system. The maker of the fabric layer essentially generates the smart contract and if the responsibility is centralized at this point to a single entity, that entity enjoys the power of controlling the whole system, given the contract and other infrastructure details are not made open-sourced. For example, the functionality to use a function in smart contract or read data depends on a mere few lines of code. If the logic behind this “access” is not made public, it can lead the players in the whole system to distrust each other.

Infrastructure Layer, a subset of fabric layer, can be made as a combination of the execution and development environment, and oracle feeds. Execution and development environment have already been discussed in the form of docker containers and EVM. Oracle feeds are outsourced data feeds that feed specific data into the smart contract in the form of transactions. For example, consider the case of a pharmaceutical supply chain which requires a specific drug to be sent via refrigerated container. An IoT (Internet of Things) device must be used which will record the temperature and send it to the blockchain. A smart contract function will be written which will get fired up in case the temperature goes below or above certain limits and enforces penalties (Celiz et al., 2018). If the oracles did not expose their logic to all the participants, and service providers were not levied penalties, it could lead to distrust yet again. Moreover, if the oracles are not properly designed, it results in garbage in – garbage out problem. Blockchain will serve no good if the data coming are faulty.

Control layer is another subset of Fabric layer which governs access to multiple parties in the system. A contract's lifecycle can be summarized into negotiations, development, deployment, maintenance and self-destruction(Wang et al., 2019). The terms and conditions defined in the negotiations, the authority and obligation of maintenance of smart contract and rules of self-destructions are coded in the respective programming language of the blockchain here. Most of the times when the smart contract is deployed, even after multiple tests, it requires updating for security and technical reasons. There are several ways to do it for example, in Ethereum it is often required to perform delegate calls via a proxy smart contract.

Finally, the smart layer also known as intelligence layer can be used to combine the abilities of artificial intelligence or machine learning into the smart contract to make it evolve and learn with the system. The vast array of opportunities for AI in blockchain can range from generating a brute force proof system to performing intelligent tasks on blockchain data. Not only smart contracts will be able to perform better due to the quality of data, smart contract inherently can be made intelligent using AI (Almasoud et al., 2018). For example, in the case of supply chain management where AI can be used for an assistive agent in defining clauses by looking at the history of the transaction from the participating accounts.

- b) **Application layer:** The smart contracts can take many forms like DApps (Distributed Applications) and DAOs (Distributed Autonomous Organizations) etc. DApps are just applications built upon blockchain, for example, Cryptokitties is a DApp on Ethereum. DAOs, rather than following the traditional hierarchical approach of organizations, are run by smart contracts with all their rules and transactions on blockchain. The design of this

application layer and the interfaces that are built upon it governs these different manifestations operations (Glaser, 2017). It is also highly dependent upon the smart layer as to what the application layer possesses. Researchers predict that using a combination of both future smart contracts will be able to perform autonomously and provide microservices like autonomous portfolio management services (Wang et al., 2019).

4.4 Smart Contracts in Supply Chain Management

Supply chains worldwide are rife with information flow and product flow inefficiencies. As already discussed in Chapter 3, the processes are extremely intertwined and complex. Moreover, parties must trust each other for veritable information and each party has an incentive to lie. Using smart contracts, the trust can be levied on technology rather than the people, which is transparent, open-sourced and robust. Smart contracts can be used in following domains in supply chains:

- a) **Paperwork:** Supply chains are multi-step echelons and require paperwork like invoices, bill of lading, letters of credit, port documents etc. at every step or link. The paperwork contains the data of the product lifecycle in the value chain as it goes downstream. With the normal process, proof of ownership of the products, to use as collateral, loan approvals and various other crucial document preparation and processes can take months. But using smart contracts, these processes can be automated and integrated with ERP fluently. For example, several large maritime carriers have joined the IBM and Maersk TradeLens Platform to reduce paperwork in shipping. By reducing the process time and the payment cycles, moving the contractual obligations and exchange payments seamlessly TradeLens aims to employ its blockchain technology across the whole shipping industry (Pope, 2019).

- b) **Tracking and Tracing:** As already discussed, products go through various transformations before turning into the final product. In the process, various product and process attributes are added. Product attributes like colour, size etc. are easier to measure without sophisticated instruments from the customer's end, but the process attributes are lost or are never captured. Using smart contracts, IoT and sophisticated hardware oracles these attributes can be measured, and rules can be enforced if conditions are not met. For example, the Toyota's venture in blockchain - in an automobile, several parts from various suppliers are assembled for the final product. To deliver the final products internationally, it requires certification of origin, change of ownership (merchanting), as well as the ability to track origin in case product recall is required due to a defect. With the use of blockchain and smart contracts the process becomes easier since all the records can be considered notarized by a distributed immutable ledger.
- c) **Supply Chain Finance:** Securities and trade settlements usually takes 2 to 3 days after the trading takes place, depending upon the countries where they are being done. Blockchains and smart contracts can considerably quicken the process by automating things and removing the need of intermediaries like custodians or clearinghouses. It can be used to make financing techniques more efficient by reducing, if not eliminating, the chances of frauds like double-counting of inventories or fake invoices. Radical approaches of finances like inventory financing will be easier and more robust which were not possible before despite the clear advantage of inventory signalling over cash signalling (Chod et al., 2019). Banks can now perform KYC much quicker, and track or trace companies on the same principle that blockchain can help to track and trace products. Projects like Eximchain and

Azhos are already popular in this space where they try to bring trade finance techniques into the blockchain space.

4.5 Choosing the Blockchain

A suitable blockchain for a specific case can be picked after understanding the requirements that are needed to be derived. This process can assist in finding whether a blockchain is even adding value to the process or not. Bitcoin, Ethereum and Litecoin are excellent use cases for peer to peer transactions. Since transactions can happen between any group of individuals public blockchain model was adopted. And due to a choice of consensus algorithm, PoW, which guarantees an exponential increase in security and robustness of the system as more “trusted” nodes join in, a public model of blockchain was the appropriate choice.

But if the information that will be shared is very sensitive, like health records or financial deals, putting them on public blockchains and making them accessible to everyone can wreak havoc. Private information and data in business are a competitive advantage for a company. For example, a classified exclusive deal with a supplier can never be shared on a public ledger. Hyperledger boasts a more scalable option which provides some added privacy. In cases that have both public and private aspects, consortium blockchains are an excellent use case. For example, Quorum by JP Morgan that was built on the top of Ethereum. Due to its restricted setting and a different consensus algorithm (PBFT), Quorum can enable several hundred transactions per second as opposed to Ethereum which can only do 15 TPS.

In an aim to develop a model which can autonomously facilitate supply chain finance, two blockchain models look appropriate: Ethereum/Ethereum-based Quorum and Hyperledger Fabric.

It must be noticed that this decision based on the current developments in the two blockchains and very much subject to change given new blockchains are evolving with added functionality. Although there have been projects like b-verify protocol to provide supply chain transparency using Bitcoin blockchain by recording the inventory transactions (Chod et al., 2019), Bitcoin is ruled out because it does not support complex scripting. As a result of the nature of problems discussed in Chapter 2, focusing specifically on supply chain finance, the interaction of the actors is hybrid. There is an interaction between B2B where an SME and a buyer will do business and a bank will be intermediary to provide purchase order and invoice financing. But at the same time, there is an element of B2C where we aim to build up a model of reverse securitization led by buyers.

For this thesis, the scope of the work will be limited to Ethereum because it provides excellent development environment IDE called Remix IDE (*Remix - Ethereum IDE*) for quick testing of a proof-of-concept. To make actual deployment and testing, Truffle Suite powered by Consensus and “Chai.js + Mocha.js” is used along with Ganache. On top of it, a full-fledged DApp can be made using libraries like REACT or ANGULAR or FLASK. Libraries like REACT uses web3 JavaScript libraries to interact with the local network on Ganache or Remix. This high integration makes development and testing of a proof-of-concept considerably easier than other blockchains. But the biggest problem with Ethereum is its permissionless platform and scalability issues in terms of transactions per second. All the transactions on Ethereum are open to public and sensitive business information can be exposed. To tackle this issue, consortium blockchains like Quorum by JP Morgan can be used. Quorum is built on top of Ethereum, so it supports all the contracts built on Solidity with an added feature of permission given by the nodes already joined. It must be

noted that Quorum uses a different consensus protocol called as QuorumChain and uses RAFT and Istanbul PBFT for fault tolerance. Since the focus of the thesis is solely on proof-of-concept, a new consensus protocol should not be of concern but calculating its impact is a top priority for the future. The only difference will be the deployment of the same contract on Quorum private network.

Hyperledger also has a strong backing of libraries developed by various developers/contributors on LINUX. It is also an equally prospective candidate but the biggest problem with Hyperledger is the unavailability of an IDE like Remix for quicker testing. Hyperledger Composer was deprecated which was previously used for quick development purpose and new IDEs like chaincoder has been developed which are in an early stage. Hyperledger has a high degree of security, scalability and flexibility but it comes at the cost of pseudo-decentralization. Using Hyperledger, the modelling of the supply chain finance system will be quite different because every supply chain will be modelled into its private network which will share data with its participants but not with anyone outside the system. Using Ethereum/Quorum, a free market can be modelled where SME suppliers, banks, buyers, carriers can all be onboarded on a single platform and do business using smart contracts and have access to different transactions using different access modifiers. Hyperledger can also support this but onboarding structure is permissioned. A study by Ernst and Young found that current private blockchain transaction cost might be 143 times less than public blockchain with zero-knowledge proof but this comparison may flip signs once the public blockchains reach their third-generation (Ernst and Young LLP, 2019). In a nutshell, whether Hyperledger or Ethereum is appropriate is subject to change in future because of consistent growth in blockchain technology.

Moreover, like all other computer technologies, there will be a high degree of integration across blockchains rather than them competing with one another. Hyperledger already has projects like Hyperledger Burrow and Hyperledger Besu which aim to use EVM to run Ethereum smart contracts. To summarize, a proof-of-concept of SCF on Ethereum's Solidity would fit perfectly for the current scope of work and quicker development, given the expected integration and cross-blockchain support in the future.

4.6 Limitations of Smart Contracts

Each smart contract's limitations depend on the blockchain on which it is deployed. For proof-of-concept, limitations will be specific to Ethereum. Based on systematic mapping the limitation comes primarily from four issues: coding issues, security issues, privacy issues, and performance issues. They arise due to bugs in programming, leaking of personal data and ability to scale (Alharby & Moorsel, 2017). The limitations can be broadly classified into two categories: contract limitations and blockchain limitations (Dika, 2017).

4.6.1 Contract Limitations

These limitations can be exploited by miners or users for personal gains and they reside in the contracts layer of the blockchain.

- a) **Transaction-Ordering Dependence (TOD)** – A block consists of multiple transactions and miner can perform TOD on a contract by tampering with the order of execution of the transactions when several of them invoke the same contract.

b) **Timestamp Dependence** – A lot of complicated functions in a smart contract use timestamps to perform operations. But miners have the freedom to set the timestamp for their mined blocks depending upon their local clock. Miners can manipulate the execution of functions which use these timestamps as triggers.

c) **Mishandled Exceptions** – Consider two contracts: a caller and a callee. If the caller doesn't check whether the callee was called properly, it can cause threats.

d) **Re-entrancy Vulnerability** – This security fault, which causes the infamous DAO attack, happens when the malicious attacker uses the fallback mechanism to reenter the caller function into the contract which calls another contract and do repeated calls. It can be used to exploit multiple refunds.

e) **Callstack Depth** – Whenever the caller contract calls another, it increments the call stack by one frame and when it reaches the limit in EVM of 1024 frames, EVM throws an exception error. A malicious actor can create a full call stack and then call the required function which throws the error.

4.6.2 Blockchain Limitations

These drawbacks come as a result of coexistence and dependency of smart contracts with blockchains and the infrastructure layer.

a) **Bugs and Performance:** Due to the immutability of the blockchain, updating smart contracts if in case a bug is detected becomes very cumbersome. For example, in Ethereum a new proxy smart contracts with delegate calls must be used. Performance issue

limitations in smart contracts reciprocate the performance limitations of blockchain which are latency to execute transactions due to consensus and scalability.

b) Oracle problem: Briefly discussed in an earlier section, the oracle problem mainly concerns with the quality of data that goes into the blockchain. Blockchain depends on external sources from outside blockchain to get the data to trigger or execute smart contracts, like web API from Binance to get the ether price. Although there have been services like Oraclize and ChainLink to provide authentic oracle solution to various blockchains, there is still a lack of development to provide oracles for every blockchain use case. If the data that goes into blockchain is not apt then blockchains with all its features render no use (Greenspan, 2016).

Chapter 5: Proof-of-Concept Smart Contract

5.1 Introduction

In this Chapter, a proof-of-concept is developed to demonstrate the use of smart contracts built on Ethereum for performing supply chain financing techniques. The smart contract(s) aims to provide the functionality to onboard the players involved in the business: buyer, seller, banks, insurance companies and investors. Using blockchain's capabilities to provide immutable information will smoothen and secure the operational flow as observed later. Three types of supply chain financing have been chosen: supplier-led purchase order financing, buyer-led invoice financing or reverse factoring and single buyer-led reverse securitization. Initial development will be done on Remix IDE by Ethereum which comes with 10 test accounts with a balance of 100 (fake) ethers each.

This Chapter is divided into three parts. A basic schematic model to perform the three financing techniques will be laid down and explained in detail along with the assumptions made in the development of the model. It will be followed by an explanation of the pseudo-code and the assumptions that have been made in the development of the code. In the end, a use case will show the test run of the smart contract on REMIX + Ganache and then a unit test will test the contract on 3 tests and 10 different subtests using Chai.js.

5.2 Supply Chain Finance Models on Blockchain

The process begins by onboarding all the players. Before describing the model, it is crucial to know what assumptions it is based on. To onboard players, each of them is assumed to have access to an Ethereum wallet where the actors can have their respective ethers stored. We also assume

that the price fluctuation of Ethereum won't cause any effect on the operations. It is understood that this assumption is unrealistic, but it can be dealt through the use of stable coins which will be explained later in Chapter 6 in the context of future directions. Figure 5.1 shows the onboarding process. A representative from each organization will put information about their company/bank to register it on the blockchain using their respective private keys. The information will contain the basic information of company like name, location, business number etc. A trusted oracle, preferably maintained by the government, will be needed to check the ownership of a company. This oracle must have data about the business and its associated public key. Only the owner of that account or the person with the private key of that company would be able to register the company with that name, address and business number. This can be done via the use of Inter Planetary File System (IPFS) and Ethereum combination or use of Hyperledger Indy where credential of ownership of the company can be given or using Quorum. A detailed explanation is provided in Chapter 6.

A similar onboarding process can be assumed for banks. Each time a registration happens, the data is uploaded on blockchain in the form of a transaction and is available for everyone to query. The ability to check whether the company exists will be extremely convenient for the banks/ investors who must perform KYC as mentioned in the previous Chapter. And such a procedure can help mitigate the financial losses and rejections due to stringent KYC guidelines (FINASTRA, 2019) and make SCF more accessible by decreasing the time required for KYC (PYMNTS, 2019). It must be noted that from this point supplier/seller will be used interchangeably.

Following this, the seller company registers the products that it wants to sell with the product name and it's selling price. In another scenario, there is an opportunity for negotiations between the

buyer and seller where the buyer sends the request for a quote and seller replies with the price demanded. But for the sake of simplicity, it can be assumed that buyer pays the price that the supplier registers that product for. In all the financing instruments, it is assumed that there are no negotiations between buyer and supplier about the price and the shipping Incoterms, and they both agree on the price specified by the supplier. The supplier/seller then manufactures the product and increments the products current inventory level, in turn, making another transaction on the blockchain. This task can be automated using a hardware oracle which measures the product quantity via IoT and then registers the product quantity on the blockchain. This way the quantity will be uploaded real-time instead of manually inputting it, which can also be prone to errors and inventory frauds.

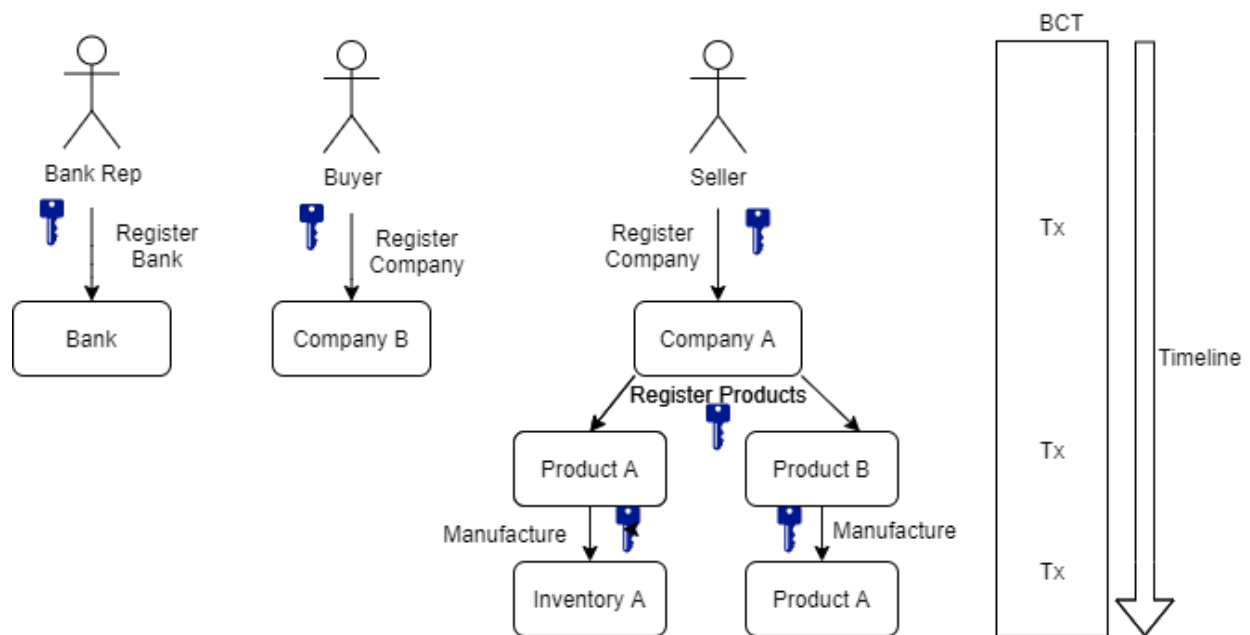


Figure 5.1 Proof -of-Concept Onboarding Process (constructed by author)

5.2.1 Purchase Order Financing

The process begins with the buyer sending a purchase request from a specific account for a product that the buyer wishes to buy. The request must contain the quantity of the product. As specified in the previous section, we will assume that the buyer agrees to buy a product at the price specified by the seller. Once the seller accepts the order specifying the carrier, due days of payment and whether invoice financing is accepted, the smart contract checks the inventory. If the quantity requested to buy is more than the inventory, the seller receives the notification to produce more. And if it is less, then the Invoice and Bill of Lading are produced, and the order is shipped. The latter scenario will be discussed in the next section. The focus here is on the prior case where inventory is lower than the quantity requested. It can be reimagined in the context that the seller is a small SME and has just received a massive order that it has insufficient capability to fulfil it immediately. If the seller specifies the ability to manufacture the product right away, the smart contract can directly facilitate with the help of a hardware oracle, but that is discussed in detail in the next Chapter. Now, if the seller is unable to fulfil the order, he/she can apply for the purchase order financing using the purchase order request ID generated by specifying the bank Id. The bank will now have access to this purchase order request data, and it can either approve it or decline it after due diligence.

Once the bank approves, the money (in form of ethers) will be sent to the seller from the bank's account by applying a specific pre-decided discount rate (80% in the current proof-of-concept) and the bank will receive tokens in return known as "REC" token which will denote the receivables that the bank bought from the supplier. Following this, the seller will use the money received to fund the operations and manufacture the product.

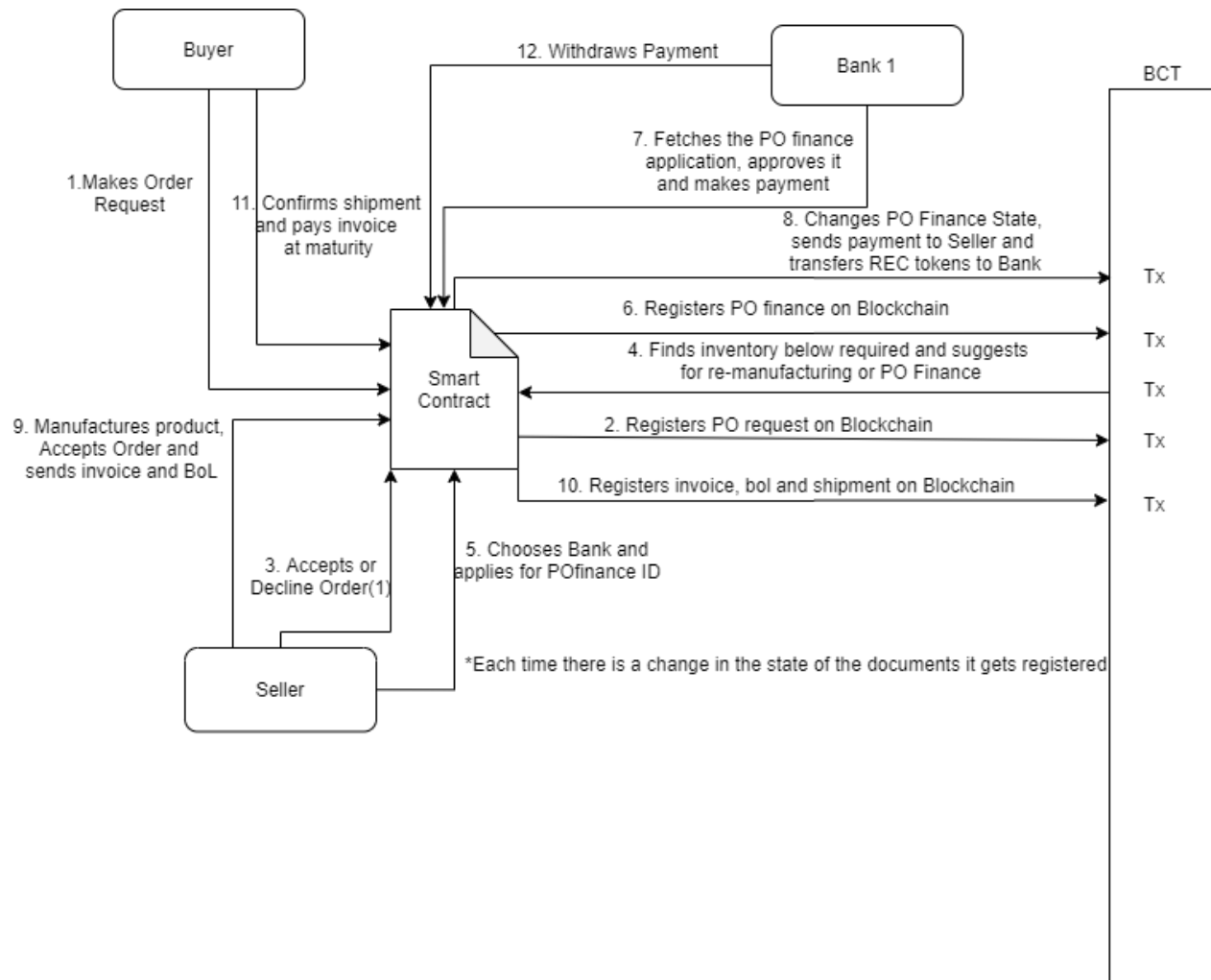


Figure 5.2 Proof-of-Concept Purchase Order Financing (constructed by author)

Once the products are manufactured, the seller again accepts the order specifying order ID, carrier ID for delivery, payment due days, and whether invoice finance is accepted. It must be noted that since the buyer has already paid via purchase order finance, invoice finance will not be accepted, even though the buyer is agreeing to accept it, to avoid the double financing fraud. This leads to the generation of invoice and bill of lading and the product is subsequently shipped. There is a need for another hardware oracle for checking the quantity shipped, quantity in the purchase order and quantity received before and after the delivery of the order to make the payment to the carrier.

For the simplicity sake, the delivery fee is not included and is a scope of future work. Once the shipment is received at the location that the buyer specified while registering the company, the buyer can confirm the shipment.

This shipment tracking and changes in the state of invoice will be accessible to the bank as well to track the progress of the order. It is assumed that shipment will reach the buyer in time without getting lost in the logistics process. Another smart contract can be used here to check the location and provenance of the shipment/order which will be discussed in the later Chapter. Once the buyer makes the payment it goes to the contract's account and the bank can now withdraw it. It must be noted that it is assumed the buyer will never default the payment and no penalties are decided if he/she does. Figure 5.2 explains the process in a schematic manner with the steps written.

5.2.2 Buyer-led Invoice Financing or Reverse Factoring

In the Chapter 2, we discussed the benefits of buyer-led financing techniques to the supplier/seller by providing a better discount rate on the early payment of invoices using buyer's credit. The process of invoice financing is almost the same until the point of generation of invoice with one big difference. In this case, like the Purchase Order finance, it is assumed that when the finance is approved, it is approved for a full order and there is no scope for partial financing. However, it is unrealistic considering partial financing is a known phenomenon. But for the developing proof-of-concept, a simple model is adopted.

Once the buyer receives the invoice and the seller of that invoice has specified that invoice finance is accepted, the buyer can choose a bank and apply for invoice finance. Banks, just like purchase order financing, will have access to purchase order, bill of lading and invoice details for a proper

check. These documents, since being on the blockchain, can be now deemed completely authentic and thus significantly reduce the risk of document frauds. Legal verification of invoice always has been one of the biggest barriers in SCF adoption.

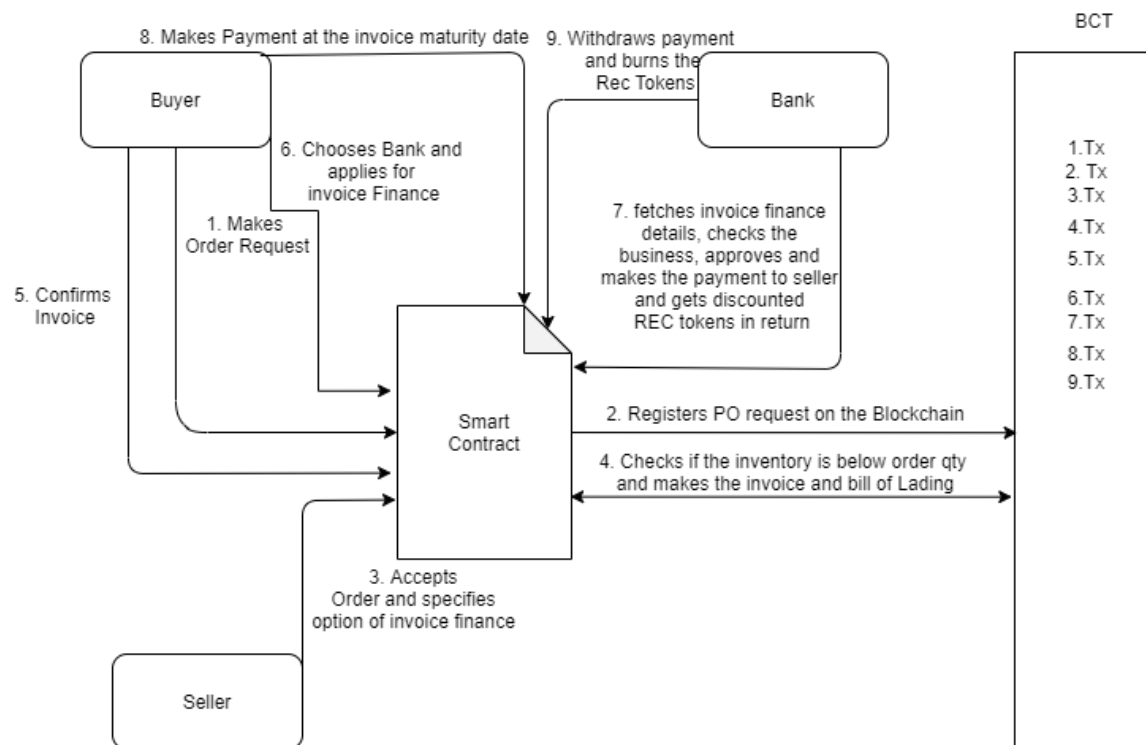


Figure 5.3 Proof-of-Concept Reverse Factoring (constructed by author)

From the buyer's side, there is a bigger assurance since chances of double payment are now exponentially low for invoices which are already sold to a third party. Moreover, banks too experience a higher degree of security if blockchains can be used to implement legal obligations on buyers who offer "promise to pay". Once the bank performs the due diligence, it approves the invoice and the money (in ethers) is sent to the seller's account with the discounted rate for invoice Finance. The bank receives the "Rec" tokens in return equal to the payment due on the invoice denoting the RPA or Receivables Purchase Agreement. At the invoice maturity, the buyer pays for

the order in form of ethers and it is deposited into the smart contract with an event on the blockchain which notifies seller to withdraw its payment. In a disruptive model, payments can be made automatic depending on several conditions which can be pre-decided or negotiated. For example, if the shipment arrives before x days of shipping then a discount will be given to the entity who is paying for the delivery or if the quantity delivered is wrong then the carrier company will be penalized. Once the payment is done by the buyer at invoice maturity, the bank can withdraw ethers by burning/using the Rec tokens. Figure 5.3 shows the steps for reverse factoring in the proof-of-concept.

5.2.3 Reverse Securitization

For the proof-of-concept, single-buyer reverse securitization is chosen because of the constraint of having at least 20 different buyers to diversify risk. There is no major structuring difference from multi-buyer structure other than the number of buyers. In this financing technique, a smart contract can expose the unpaid invoices owed by a buyer to the open market where it can be bought by investors and maintained by their custodians. The process is the same as the previous techniques to the point where the buyer receives an invoice from the seller who specifies whether the invoice finance is accepted or not. When the buyer has one or more than one unpaid invoice eligible for finance, he/she is eligible to apply for reverse securitization. It is assumed that the point of time at which buyer applies for reverse securitization, he/she aims to securitize all the unpaid invoices accepting invoice finance. To apply for reverse securitization, the buyer will pick the insurance company which will rate the SPV (special purpose vehicle) generated in the process. The responsibility of the insurance company is limited to rating only, but it can be expanded as shown in Chapter 6. Moreover, different models can be adopted here to choose the insurance company,

where instead of the buyer, investors can choose their insurance provider for the securities they are buying. Registering or onboarding the insurance company on the blockchain is the same process as any other company. The insurance company gives a rating on the scale of 1 to 5 with 5 analogous to AAA rating. This scale is chosen due to the nature of its simpler understanding.

Once the buyer applies for reverse securitization after picking the insurance company, it registers the detail of the securitization on blockchain and the smart contract creates the SPV or Special Purpose Vehicle to facilitate the securitization process. The SPV's address is the same as the contract address. Although this can have legal implications since the contract address is maintained by the deployer, it is assumed that for proof-of-concept this problem is out of scope.

The investor examines the security issued by evaluating the SPV's rating and decides to buy it. To purchase the security, investor inputs the amount of money (in ether) with which it wants to buy the security and the security ID. For the calculation purpose, it is assumed that the discounted rate at which investors purchase the security will be the same as the invoice discount. On a successful transaction, the investors receive "SEC" tokens along with the notes which contain the information of the transaction in their wallet. There is no need for clearing or settlement providers in this model because blockchain plays that role as a distributed immutable ledger. This also significantly reduces the settlement period from days to almost minutes.

However, there is also scope for a less disruptive model which will keep the clearing providers in the process. For example, instead of deploying a global peer to peer solution, smart contracts/blockchain technology can be directly adopted by settlement providers without any reforms in the infrastructure. It can be deployed only for the second market while the primary markets still work

the same. Once the investor buys the security, the money (ethers) is deposited in the smart contract and sellers are notified. Sellers can withdraw the money from the smart contract by burning the REC token they received when the buyer confirmed the invoices. The withdrawal is based on the first come and first serve basis.

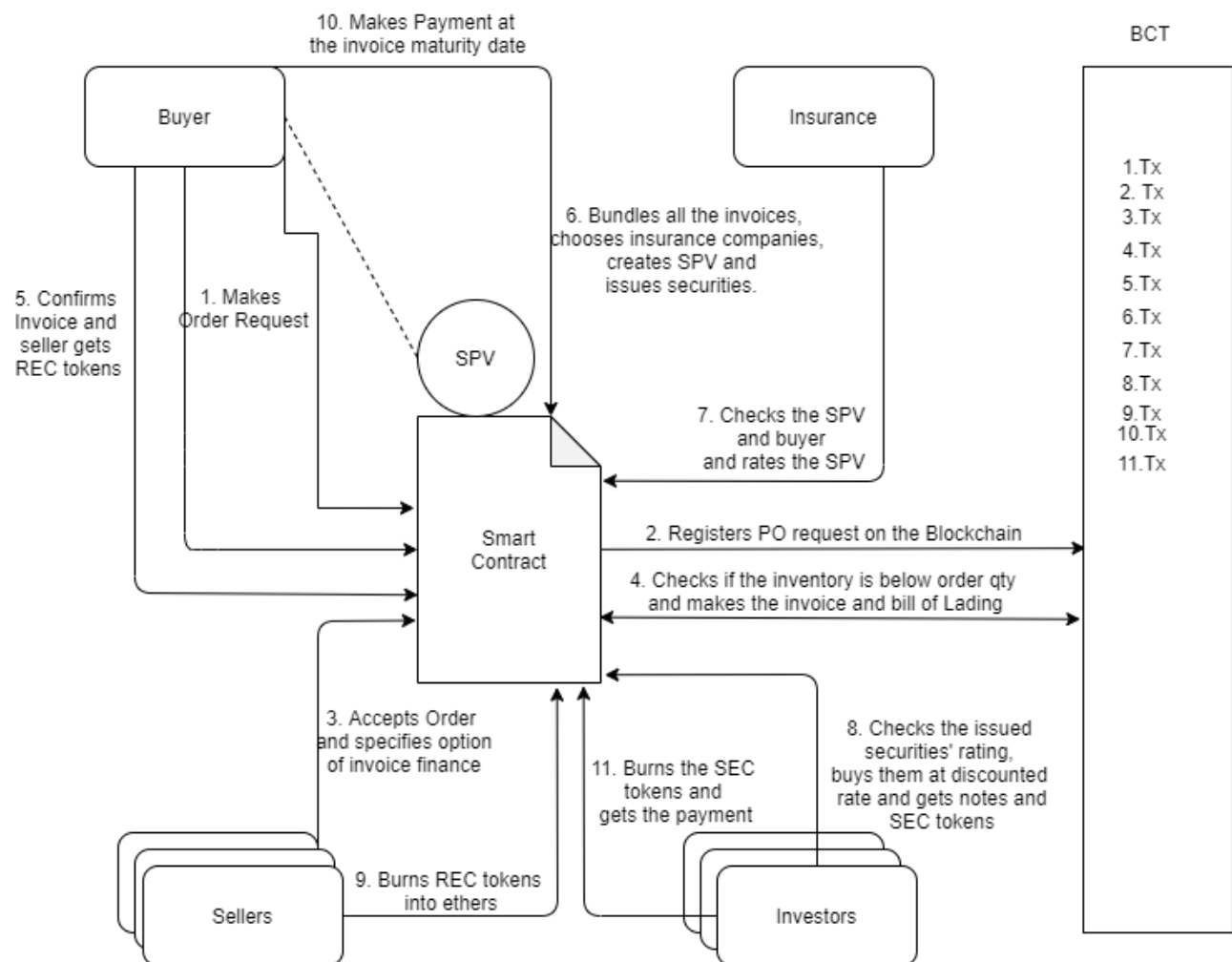


Figure 5.4 Proof-of-Concept Reverse Securitization (constructed by author)

One major difference in this model of BCT based reverse securitization is calculating all the payables by the buyer to all the sellers that accept invoice finance and bundling them up. This

bundle is immutable once this bundle is issued as a security. In the reverse securitization model depicted in Chapter 2, buyers first upload the invoices on the technology platform and then suppliers request early payment later which can trigger the creation of SPV which in turn issues the information on a CSD. But in this model, sellers specify whether they accept invoice finance at the instant they accept the order and hence the bundling of invoices precedes the making of SPV's. Although this removes some of the flexibility for the sellers, it speeds up the process. After an assessment of this model versus the old securitization practice, an informed choice will be made in the future. Moreover, due to the typical nature of Ethereum blockchain which requires gas for every transaction, the latter process appears to be more optimized. If the requirement is to follow the same process, the same SPV will be updated every time a new request for early payment is done. And to make this work, it is convenient to deploy all the financing techniques in their respective separate smart contracts which can interact with each other by function calls, as discussed in detail in the next Chapter.

In any securitization process, there are two options: with and without recourse. We assume that securities issued are without recourse based on the assumption that buyers will never default hence the limited role of insurance companies can be justified. Once the buyer makes payment or in a disruptive scenario, payment can be automatically withdrawn from the buyer's account, at invoice maturity investors are notified. The investors then withdraw their money based on first come first serve basis simultaneously burning their SEC tokens.

Lastly, another assumption here is the exclusion of banks in the process. In a real-world scenario, banks play a crucial role in buying receivables/payables from the buyer or seller, depending on the

nature of securitization, and then securitize the invoices. But due to gas constraints in the smart contract, the current model is used, and a more realistic contract will be developed in the future.

5.3 Pseudocode

The proof-of-concept demonstrated by smart contracts in **appendix 1 and appendix 2**, contract's interface in **appendix 3** and the "SafeMath" library shown in **appendix 4**, was developed on Solidity. The compiler range of 0.5.0 to 0.6.9 (preferably 0.5.17+) can be chosen to compile and deploy the Solidity code. For development purposes of proof-of-concept and quick testing of the code, remix IDE will be used. It must be noted that due to the size of the contract, it is required to "enable optimization" and increase the "gas limit" to 30,00,000,000, which is "3,000,000" by default. The code will be optimized in the future to reduce the "gas cost". Before delving deeper into the actual code, it is important to know a few basic things about Solidity. In addition to basic data types like uint (unassigned integers), string and bool, Solidity has a custom data type called as a "struct" which can group various data types into an object. For example, the following struct "Person" is made up of two strings and an uint. It must be noted that if the uint's size is not specified then it becomes uint256 by default. Other options are uint8, uint32, uint64 and uint128.

```
struct Person {string name;
```

```
    string address;
```

```
    uint weight;}
```

Solidity also supports events. When events are emitted in a function call, they store the arguments in the transaction logs of the blockchain. Events are only accessible outside the contract,

specifically mostly on the front end and can be used by specifying the contract address. Solidity also provides usage of modifiers which enables additional restrictions on the functions. Instead of repeating the same code in every function where the same restriction is required, modifiers enable code reusability. Given below is the most common modifier which restricts the access to call the function to only the owner of the contract.

```
modifier onlyOwner () {
```

```
    require(msg.sender == owner);
```

```
    _;
```

```
}
```

Solidity has a unique functionality called mapping which acts as hash tables with key and value pairs. Mapping is used to make associations. For example, the mapping below stores the tokens for every address.

```
mapping(address => uint) public ownedTokens;
```

Solidity smart contracts support object-oriented programming and hence it supports inheritance, polymorphism and encapsulation. Inheritance denotes the capability of contracts to develop a child-parent relationship to prevent code-redundancy. Contracts can inherit functions variables, modifiers and events. This is done by copying the bytecode of the base/parent contract into the child contract. Solidity supports single, multiple and hierarchical inheritances. Solidity also allows two kinds of polymorphism: function and contract. Function polymorphism denotes multiple functions in same or inherited contracts with the same name but different parameters. It is also

called method overloading. Contract polymorphism refers to using multiple contract instances. It can be used to trigger derived contract functions.

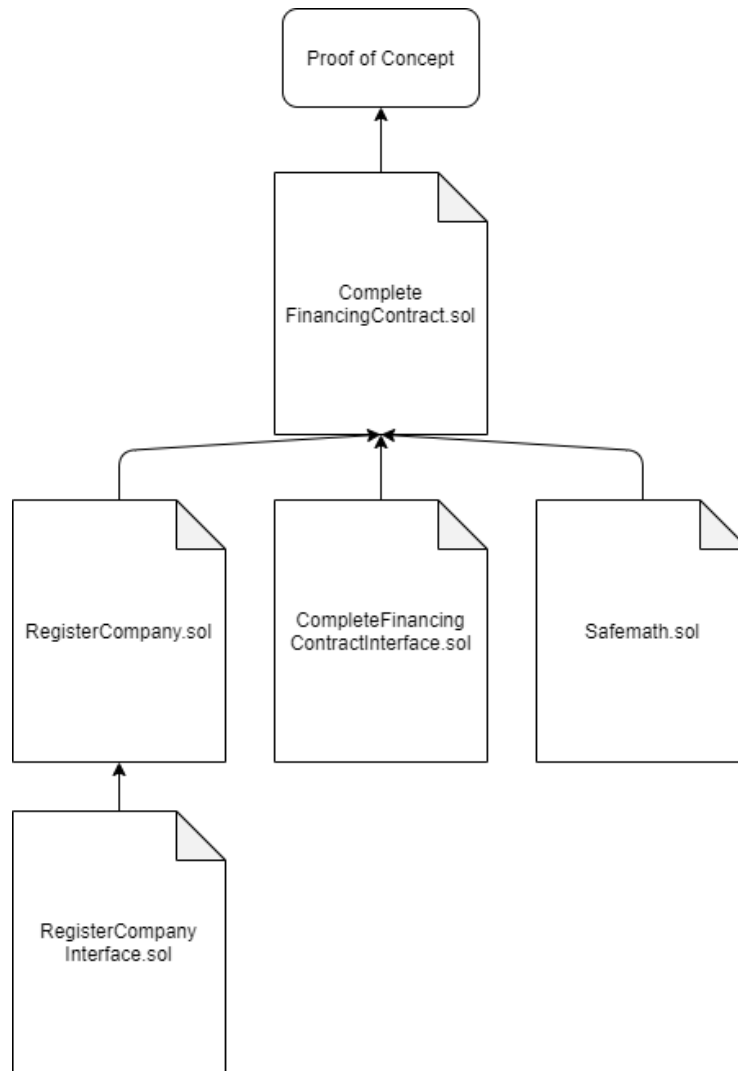


Figure 5.5 Smart Contract Composition (constructed by Author)

Solidity also offers encapsulation which refers to restricting the access to state variables and functions to a specific or a group of clients. Solidity offers the following visibility modifiers: external, internal, private, view, pure, public. Here external functions and variables are only accessible outside the contract and vice-versa for internal and private. The difference between

“private” and “internal” is that in “internal” visibility, functions and state variables are accessible inside the derived or child class as well but this feature is not available with “private” visibility which restricts it to the contract where it was specified. “View” functions can read but cannot modify the state variables while “pure” functions can neither read nor modify the state of the variables.

Another peculiar feature of a Solidity smart contract is the use of interfaces. Appendices C and D shows the interface for the smart contract, which is an abstract contract containing undefined external functions which are accessible only outside the contract.

Figure 5.5 shows the composition of the smart contracts developed as a proof-of-concept. It is named as “CompleteFinancingContract.sol” which has the required codes for performing a normal business operation without financing, purchase order financing, reverse factoring and reverse securitization. It is derived from the parent contract “RegisterCompany.sol”, imports a library “SafeMath.sol” and has the contract interface “ContractInterface.sol”. The details of each pseudocode will be explained in further subsections.

5.3.1 RegisterCompany.sol

Shown in **appendix A**, this contract has a constructor method which defines the address which deploys the contract as admin address. This address can be later used to perform admin specific tasks. The contract has two structs for company and product. Company is made up of company ID, company’s name, business number, location, owner’s public address and bool which specifies whether it is an insurance company or not. The “*Product*” struct is comprised of product’s ID, product’s name, company ID which has registered the product and selling price of the product.

There are two private unassigned integers defined for ID's which will increment every time a new company or product is registered respectively, as shown further. Two internal uint *"totalcompanies"* and *"totalProducts"* keeps track of total companies and products registered. These can be used to loop over the companies and product in the child contract. There are three mappings defined out of which *"companies"* and *"products"* are public or in other words, they can be called by anyone on the network to find their details by putting the index of the respective mapping. *"ownedCompanies"* is a private mapping which keeps track of the total number of companies owned by an address. There are two events which are emitted when companies and products are registered. Following are the functions used in "RegisterCompany.sol".

- a) **createCompany**: This function takes *companyName*, *location*, *businessNumber* as an input to register the company on the blockchain. But to have a successful transaction, it is required to not have empty strings while calling the function or the transaction is reverted. On successful transaction, the company gets appended to the mapping *"companies"* simultaneously incrementing *companyID*, the total company registered, and companies owned by the calling address.
- b) **getCompany**: This public view function takes *companyId* as an input and gives detail about the respective company.
- c) **registerProduct**: This function takes the company ID, product's name and selling price as an input. If the addresses are not allowed to have more than one company ID, the first input can be neglected, but in the proof-of-concept, one address can have as many companies registered to it. On successful calling of the function, the product gets appended to *"products"* mapping and increments the product ID and total products registered.

- d) **getProduct**: This public view function takes *productId* as an input and gives detail about the respective product.

5.3.2 Contract Interfaces

Appendix C and D contain the interface for the contract. An interface is an abstract contract filled with function declaration which can be used for the front-end development. The functions included are the same as the one found in “RegisterCompany.sol” and “CompleteFinancingContract.sol” with “external” keywords, demonstrating “polymorphism” in Solidity. Each of the smart contracts has a different interface associated with it which is imported in its respective Solidity file. A separate interface is required because if a single interface is made and all the functions of those interfaces are not used by the importing contract, the importing contract cannot be deployed because it is considered as an abstract contract by the Solidity compiler.

5.3.3 SafeMath.sol

To understand the importance of “SafeMath” library, it is crucial to understand the concept of overflow and underflow. Every “uint” or unsigned integer has a maximum and minimum limit. For example, the maximum number uint8 can store is $2^8 - 1$ or 255 and if it is incremented above that, it turns into 0 instead of 256. Similarly, the lowest number that can be stored is 0 and if 1 is subtracted from 0, uint8 will turn into 255. This former phenomenon is known as overflow and the latter is known as underflow. Safemath’s library shown in appendix 4 helps to prevent mistakes in codes caused by these phenomena.

5.3.4 CompleteFinancingContract.sol

This contract begins by importing “SafeMath”, “RegisterCompany” and the “ContractInterface”. “SafeMath” library is described elaborately in the next section. This contract has a constructor which defines the variable “*contractAddress*” as the address of the contract. To understand the contract easily, it is suggested to divide the contract into four virtual partitions: business, purchase-order financing, invoice financing/reverse factoring and reverse securitization. **Appendix F** gives a quick summary of the sectioning of the structs, variables and functions.

The “business” part of the contract contains structs for “*PurchaseOrderRequest*”, “*Invoice*”, “*Shipment*”, “*BillofLading*”. It contains the mappings “*purchaseOrderRequests*” which stores all the purchase requests, “*productToInventory*” which stores the inventory of a product and “*orderId*” which increments whenever an order is placed. “*totalbolId*”, “*totalInvoices*”, “*trackingId*” as private variables which stores IDs for bill of Lading, invoices and shipment respectively. The functions “*manufactureProd*”, “*getInventory*”, “*removeProduct*”, “*approveReceivables*”, “*approveRecievables*”, “*makePurchaseOrderRequest*”, “*getPurchaseOrderRequests*”, “*acceptOrder*”, “*makeInvoice*”, “*makeBillOfLading*” belong to this section. The details of structs and variables can be found commented in the **appendix B**. The functions are explained below:

- a) **manufactureProd**: When the seller wants to update the inventory, this function must be called with inputs “*_productId*” and “*_amount*” and it updates the inventory mapping of the inputted *product's* “*productToInventory*” by quantity equals to “*_amount*”. It requires that only the address that has been registered to that product can update it. There is an

opportunity of a hardware oracle, which automatically updates the product's inventory on creation by firing this function.

- b) **getInventory**: This is a view function which enables only the product's owner to view the inventory. It doesn't cause any change in the state.
- c) **removeProduct**: Although this function appears to have more relevance in "RegisterCompany", it operates more efficiently in "business" because this function has to have access to "*purchaseOrderRequests*" and "*invoices*" mappings. If the product's order is accepted it cannot be deleted until the order is fulfilled. Otherwise, the owner of the product can delete it.
- d) **approveReceivables**: This function takes *quantity* ordered and price of the product and calculates the total receivables earned by the seller using SafeMath's "*mul*" function.
- e) **approveRecievables**: This function fires up in other functions when there is a requirement to approve receivables from one entity to another. For example, when the bank approves invoice finance the receivables are allotted to the bank from the buyer. It increments the allowance mapping from buyer's address to bank's address by the specific amount.
- f) **makePurchaseOrderRequest**: This function takes the buyer company's Id, the product's ID which must be purchased, and the quantity required as an input and generates a "*PurchaseOrderRequest*" by incrementing the "*orderId*" and appends it to "*purchaseOrderRequests*" mapping. While generating it marks "*isAccepted*" and "*isFinanceRequested*" as false by default. To execute this function, it is required that the "*address*" should not buy its own product and "*address*" should be the owner of the "*buyerCompanyId*".

- g) **getPurchaseOrderRequest:** By calling this function, the buyer company, the seller company and if the finance has applied to a specific bank, then only that bank, can pull the information of purchase order's status by putting in the order's ID.
- h) **makeInvoice:** This function is a private function and can only be called by a function in the smart contract. On the successful function call, this function generates an "Invoice" and appends it to "invoices" mapping incrementing the "totalInvoices". It takes the "orderId", due days of payment and whether invoice finance is accepted as an input, and if the "invoiceFinanceAccepted" is marked as true, then it increments "companyOwnedFinanceAcceptedInvoices" by one.
- i) **makeBillofLading:** This function takes "orderId" and "carrierId" as an input and generates a "BillofLading", further appending it to "billofLadings" mapping. This is also a private function. Although "BillofLading" does not resemble the physical bill of lading, the changes can be easily made by making changes in the struct "BillofLading".
- j) **shipThisOrder:** This is also a private function which is called when the order is accepted. It takes "carrierId" and "orderId" as an input and generates a "shipment" and appends it to "orderToShipment" mapping. This function takes the addresses from both companies in business and initiates the shipment from seller's address to the buyer company's address.
- k) **getInvoice:** This is a view function which enables the buyer company, the seller company and the bank, if finance is requested, to view updates on *invoice* by inputting the "orderId".
- l) **getBillofLading:** This is a view function that enables the buyer company, the seller company and the bank, if finance is requested, to view updates on the bill of Lading by inputting the "orderId".

- m) **getShipment:** This is a view function that enables the buyer company, the seller company and the bank, if finance is requested, to view updates and details on shipment by inputting the “*orderId*”.
- n) **acceptOrder:** This function can only be executed by the seller who has received a purchase order Request. To accept the order, the seller inputs the “*orderId*”, the “*carrierId*” which will transport the order, “*paymentDueDays*” and whether he/she accepts the invoice finance. On the execution of the function, the function checks whether the inventory of the product is more than the order quantity. If it is, then three functions: “*makeInvoice*”, “*makeBillofLading*” and “*shipThisOrder*” are called, and “*OrderAccepted*” event is emitted. If the quantity is less, a failure event is emitted which signals the seller to manufacture more product or apply for purchase Order Finance. If the seller has applied for purchase order finance, then even if he/she specifies that invoice finance is accepted, “*isInvoiceFinanceAccepted*” is changed to false to prevent double payment to the seller.
- o) **confirmShipment:** Only the buyer can execute this function by inputting the *invoiceId* to confirm that the shipment has been delivered and the invoice has been approved. This function has a series of if-else statements which perform various checks and transfer “Rec” tokens accordingly. If purchase order Finance is requested then, it transfers the REC tokens to the bank equal to the receivables and marks that the seller has been paid by the bank. If there has been no finance then it allots the REC tokens to the seller who can liquidate them using “*LiquidateReceivables*” function.
- p) **cancelOrder:** If the order is not accepted then the buyer can use this function to cancel and delete the order details from the “*purchaseOrderRequests*” mapping. It doesn’t delete

anything from the blockchain, instead it makes another transaction on the blockchain which depicts deletion of an order.

- q) **makePayment:** This payable function is used by the buyer to make the payment of the invoice. It takes “*invoiceId*” as an input and requires the buyer’s balance to be more than the payment required.
- r) **LiquidateReceivables:** This function can be used by the suppliers, banks and any other “address” to withdraw money from the smart contract while simultaneously burning the “Rec” tokens.

The purchase-order financing part contains struct for “*bank*”, “*PoFinance*”. It uses following mappings: “*poFinances*” which stores all the purchase Order Finances, “*banks*” which stores all the bank’s details, “*isPOFinanceApplied*” keeps track whether PO finance was applied for the supplier (this mapping prevents the “buyer” from knowing whether financing is being requested for a specific “*orderId*”) and “*ownedbanks*” which denotes the total number of banks owned by an address. In addition to this, it uses private variable “*poFinanceId*” to keep track of purchase order finance IDs and a public variable “*poDiscountRate*” equals 80 *percentwhich* denotes the discount provided on purchase order Finance. It also contains the following functions:

- a) **registerBank:** This function is used to onboard bank on the blockchain. This function requires the bank’s name and the bank’s identification number. There is a clear need for a software oracle here which will check the bank’s public address and identification number to the one in database and only that private address that has access to the public address will be able to onboard the bank. On successful execution, the function increments the “*bankId*” and appends the “*Bank*” into “*banks*” mapping.

- b) **getBank:** This function can be used by anyone to see the information of bank by inputting bank's ID.
- c) **applyPoFinance:** This function is used to apply for purchase order finance if the inventory is below the order quantity. It requires "*orderId*" and "*bankId*" as an input and can only be applied by the seller, given he/she has not already applied for any finance and document mappings have not been generated. On successful execution, it changes the "*isFinanceRequested*" state to true and generates a "*PoFinance*" incrementing the "*pofID*" and appends it to the "*poFinances*".
- d) **getpofId:** This view function enables the seller and the bank to look at the details and the update on "*PoFinance*" state.
- e) **approvePOFinance:** Once "*POFinance*" is generated, the respective bank can approve and send the money to the seller at "*poDiscountRate*" using this payable function. The receivables are calculated using the internal function "*approveReceivables*" from the business part and then sent to the seller. Once the money (in ethers) is sent successfully, banks receive "*Rec*" tokens in the *recAllowances*" mapping. It has to be noted that "*poFinance*" state for "*isApproved*" is changed to "True" so that the bank does not resend the money to the same "*poFinance*". In the end, an approval of "*poFinance*" event is emitted.

It must be noted that the same struct for the bank will be used in "*invoice financing*" part, in addition to struct "*InvoiceFinance*". This section using the following mappings: "*invoiceFinances*" which contains all the successfully applied invoice finances, "*companyOwnedFinanceAcceptedInvoices*" which denotes the number of invoices that accepts

invoice finances owned by a specific company. It contains a private variable “*invoiceFinanceId*” which keeps track of the invoice finance IDs and a public variable “*invoiceDiscountRate*” which declares the discount given on early payment. It encapsulates following functions: “*applyApprovedPayableInvoiceFinance*”, “*getInvoiceFinanceId*”, “*getInvoiceFinanceStruct*”, “*approveInvoiceFinance*”, “*getInvoiceFinance*”.

- a) **applyApprovedPayableInvoiceFinance**: This function can only be executed by the buyer of the product by inputting “*invoiceId*” and “*bankId*” if invoice finance or purchase order finance has already been not requested by the buyer. It required that the invoice must be approved by the buyer before applying for the invoice finance and seller of that invoice must accept the invoice finance. On successful execution, “*totalInvoiceFinances*” increments by one and “*InvoiceFinance*” is generated appending it to “*invoiceFinances*” mapping.
- b) **getInvoiceFinanceId**: This is a private function which is called in another function to get the ID of invoice finance by inputting the *invoiceID*.
- c) **getInvoiceFinanceStruct**: This is an intermediate function which spits out the struct of “invoice finance” by inputting the “invoice financed”. This function can only be called by the bank, seller and the buyer. At first glance, it appears to be redundant, but it serves the purpose to reduce the call stack depth of the “getInvoiceFinance” function.
- d) **approveInvoiceFinance**: This payable function can only be executed by the respective bank to approve the invoice finance applied by the buyer. On successful execution, it sends the money (in ethers) at a discounted rate to the seller and in return, the bank gets the “Rec”

tokens equal to the receivables form the order. At the execution, it emits an event of the invoice finance approval.

- e) **getInvoiceFinance**: on the successful execution of this view function, the bank, the seller and the buyer can fetch the details of the invoice finance by inputting the invoice finance ID.

Lastly, the remaining structs for “*Spv*”, “*Security*” and “*Note*” belong to “*Reverse Securitization*” partition. It also contains mappings of all these structs which stores information about all the “*spvs*”, “*securities*” and “*notes*”. This section uses private variables: “*sID*” which denotes the ID of the security and total securities issued, “*spvID*” denotes the ID of the SPV and total SPVs generated, “*noteId*” denotes the note’s unique ID once bought by the investor, and a public variable “*invoiceDiscountRate*” which denotes the discounted percentage if investors buy the security. It is same as “*invoiceDiscountRate*” in Reverse Factoring for simplicity purposes. To enable the process, this section uses the following functions:

- a) **registerAsInsuranceCompany**: This function operates similarly to “*createCompany*” function from the “*business*”. The only difference is that unlike the “*createCompany*”, on the execution of this function by inputting “*companyName*”, “*location*” and “*businessNumber*”, a new “*company*” is generated with bool “*isInsuranceCompany*” as true and subsequently appended in “*companies*” mapping.
- b) **createSpv**: This function creates an SPV which will be rated by the insurance company. This is a private function and will be called in “*applyReverseSecuritization*” to generate an SPV. This function takes the security issuer’s ID and insurance company’s ID as inputs

and generates an “SPV” appending it to “spvs” mapping. The “SPV” generated will have a default zero rating.

- c) **rateSpv:** Only the insurance company which was specified while creating the SPV can call this function. It requires the insurance company to input the “spvId” and the “rating” it wants to give to the SPV. The successful execution changes the state with the new “rating”.
- d) **getAllBuyerInvoiceFinancePayable:** This is a private function which takes “buyerCompanyId” or securities issuer’s ID as an input and returns a “uint” which is the total payable from all the invoices that accept invoice finance that issuer promises.
- e) **applyReverseSecuritization:** This public function can be used by the buyer to securitize all the invoices accepting invoice finance. It takes two inputs: the ID of the company that wants to securitize the invoices and the insurance company ID which will rate the SPV in the process. This function calls two private functions to calculate the total payable that the buyer promises and subsequently creates an SPV. The function generates a “Security” struct and appends it to “securities”.
- f) **buySecurities:** This function can be used by investors or their custodians to buy securities by inputting the security ID. Hence this is a public payable function. On the successful execution, “noteId” is incremented by one and a new “Note” is generated. This “Note” is then appended to “notes” mapping. It also allots “Sec” tokens to the investor by incrementing “secAllowances”.
- g) **getNoteInfo:** This is a public view function which can be used by the investor to only see details of their respective note that has been owned by that “address”.
- h) **liquidateNote:** Once the buyer makes the payment for the invoice, investors can use this function to withdraw the ether into their account from the smart contract address by

inputting the *noteId*. At the successful execution, the note's Boolean *isRedeemed* is marked as true so that it cannot be used again.

5.4 Use Cases

For quick testing purposes an inbuilt UI is provided by "REMIX". The smart contract will be deployed on "GANACHE CLI" using "REMIX IDE". GANACHE enables deployment of Ethereum blockchain on local nodes and it connects to REMIX using a "web3Provider". Four use cases will be demonstrated: no-financing, purchase order financing, invoice financing or reverse factoring and reverse securitization. More comprehensive testing will be done in the next section using a JavaScript testing file while deploying and migrating the contract on local node using "Ganache" and Truffle Suite. It must be noted that the smart contracts currently do not use any Oracles and they will be discussed under future work in the next Chapter. Img G.1 given in appendix G shows the compilation details and Img G.2 shows 8 of the 10 test accounts with 100 ethers (fake) each. Img G.3 demonstrates the contract creation after deployment from account 1.

5.4.1 Use Case 1: No - Finance

First, let's onboard all the players: buyer, supplier1, supplier2, logistics company, bank and insurance company using the first six accounts and remaining 2 accounts will be used as investors. Using the functions mentioned above, supplier1 will be created as (supplier, supplier,1); buyer will be created as (Buyer, Buyer,2); the logistics company will be created as (carrier, carrier,3); the bank will be created as (Bank, BankCode); the insurance company will be created as (Insurance, Insurance,4); and supplier 2 will be created as (supplier2,supplier2,5). Taking supplier 1 as an

example, the name of the company is supplier 1, location is “supplier1” to avoid confusion and the business number is “1” for reducing complexity. Img G.4 shows the onboarding transactions.

Now using “supplier” and “supplier 2”, let’s register two products (1,product1,10000000000000000000), (6,product2,10000000000000000000) using “*registerProduct*”. Companies and products can be seen using “*getCompany*” and “*getProduct*” as shown in Img G.5 and Img G.6. For use case 1,2 and 3, only supplier 1 will be needed. Using “*manufactureProd*”, “supplier1” updates the inventory by 2. It can be checked by “supplier1” using “*getInventory*” as shown in Img G.7. Switching to “*buyer*” account, a purchase request (1,2,2) is made using “*makePurchaseRequest*” for two “*product 1*” which leads to the generation of the invoice when “*supplier*” executes “*acceptOrder*” with input (1,3,30, true). Img G.8 shows the purchase order, invoice, bill of lading and the shipment. The buyer executes “*confirmShipment*” and the supplier gets 2000000000000000000 “REC” tokens as shown in Img G.9. The “supplier” then withdraws the ether as shown in the figure once the buyer makes the payment by inputting invoice ID. Img G.10 and Img G.11 shows the transaction in the ganache for payment and withdrawal and newly updated balances in both accounts.

5.4.2 Use Case 2: Purchase Order Finance

The “*buyer*” makes another purchase request for “*product1*” but when the “*supplier*” accepts, it produces neither the invoice nor any other document structs because there is sublevel inventory, as shown in Img G.12. The “*supplier*” then applies for purchase order which is accepted by the “*Bank*” and supplier receives money (ether) instantly after applying the PO discount rate as shown in Img G.13. The “*supplier*” then manufactures the “*product*” again and updates inventory by 10.

He accepts the order and the buyer receives the shipment with the invoice and the bill of lading. When the buyer confirms the shipment, the bank receives the REC tokens. Once the buyer makes the payment, the bank can cash it using “*LiquidateReceivables*” as shown in Img G.14.

5.4.3 Use Case 3: Invoice Finance

For this use case, the buyer makes the order for 1 unit of “product1” and the process will be the same until the point when the supplier accepts the order marking “*isInvoiceFinanceAccepted*” as “*true*” and sends it to the “*buyer*”. The buyer confirms the shipment, approves the invoice and then applies for reverse factoring via “*applyApprovedPayableInvoiceFinance*”. The Img G.15 shows the invoice finance information. “*Bank*” performs the due diligence via “*view*” functions and approves the “*invoiceFinance*” which deposits the “*REC*” tokens in its account and sends the money to the supplier with 90% discount of invoice financing rate as shown in Img G.16 Following this, the “*buyer*” makes the payment and bank can withdraw the money (ethers) by burning “*REC*” tokens, similar to the previous use case.

5.4.4 Use Case 4: Reverse Securitization

In this use case, two suppliers will be used to demonstrate the calculation of total payable for all the invoices that accept invoice finance, that is specific to the “*buyer*”. The “*buyer*” makes purchase request to both the suppliers for 2 units of “*product1*” and 2 units of “*product2*”. They both accept the order and send invoices specifying the option of invoice finance. The buyer then confirms both the invoices and applies for reverse securitization by choosing the insurance company. Img G.17 demonstrates the generated security with total payable of “40000000000000000000” and SPV ID “*I*”. The insurance company then rates the SPV out of 5.

Img G.17 also shows the rated SPV. The investors can perform due diligence use “*getSPV*” to find the rating, buy the security and get notes in return. Img G.18 shows the notes 1 and 2 owned by two investors. It must be noted that when investor 1 bought securities worth 2 ethers, it received 22222222222222222222 “*SEC*” tokens in return, and when investor 2 tried to buy securities of 2 ethers later, it received all the remaining 177777777777777778 “*SEC*” tokens after debiting 1.6 ethers. Their money is deposited in the SPV’s address which is same as the smart contract’s address. The “*supplier*” and “*supplier2*” can withdraw this money by burning the “*REC*” tokens they have in their account, which they received when “*buyer*” confirmed their shipments. Img G.19 demonstrates that 1.8 ethers are deposited in both suppliers account at a discounted rate of 90% because they had 2×10^{18} tokens each. When the buyer makes the payment of 4 ethers for both the invoices, investors can burn their “*SEC*” tokens for 1 Wei each (1 Ether = 10^{18} Wei) and the resulting balance is given in the Img G.20.

5.5 Testing Smart Contract

Before deploying any application, it is indisputable to thoroughly test it. The same idea also applies to test the Ethereum smart contract. Another advantage of Solidity and Ethereum is the tools developed around them to facilitate its development and growth. Truffle Suite by Consensys, deemed as the “Ethereum’s swiss army knife”, has been at the very core for deploying and testing the contracts in Solidity. Truffle can be used alongside JavaScript testing library “Mocha.js” and “Chai.js” to apply various assertions for testing purposes. The JavaScript test file “*CompleteFinancingTest.js*” in Appendix H contains the tests for testing the proper functioning of the smart contract.

Instead of delving deeper into every line of code, this section will focus on giving the holistic picture of tests done and their results. The first step requires installing all the dependencies (Node.js, Truffle suite, chai, chai-as-promised). To enable the testing, it is required to start the project by “*truffle init*”. And then the smart contracts must be put in the contract’s directory. The “*truffle-config*” file must be changed to enable the optimization and choose the network. A second migration file must be made in the “*migrations*” directory for migrating the smart contracts. Following this, “*truffle migrate*” command will compile and migrate the contract. The test file must be put in the “*test*” folder and “*truffle test*” will run the test file.

```
buzzlite@buzzlite-HP-Laptop-14-cf0xxx:~/Documents/Thesis$ truffle test
Using network 'development'.

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

Contract: CompleteFinancingContract
CORRECT DEPLOYMENT.
  ✓ should return Contract Address
  ✓ should return correct token names (49ms)
BASIC BUSINEE OPERATIONS MUST BE FLUID
MUST ONBOARD ACTORS AND PRODUCTS CORRECTLY.
  ✓ should onboard suppliers correctly ---> createCompany and GetCompany must work (188ms)
  ✓ should allow suppliers to register product and update inventory (647ms)
  ✓ should onboard Bank and Insurance Company (290ms)
MUST ENABLE ORDERING, ACCEPTING AND PAYMENT
  ✓ should be able to make order request and accept it (806ms)
  ✓ buyer should be able to make payment and seller should be able to accept the payment (579ms)
FINANCING SHOULD WORK
  ✓ seller should be able to apply and bank should be able to accept POF (1002ms)
  ✓ should allow buyer to apply invoiceFinance and bank to accept it (1615ms)
  ✓ should allow buyer to securitize the invoices and investors to buy the securities issued (3229ms)

10 passing (9s)
```

Figure 5.6 Unit Test Results of the Smart Contract

The test files were made of 3 tests which included a few sub-tests for their completion. Correct deployment of the contract checked the contract’s address and the name of the two “*REC*” and “*SEC*” tokens. Test 2 tested that the basic operations must be fluid depending upon whether all the

players of the network are onboarded correctly: suppliers can register their products and update their inventory, buyers can place the order and suppliers can accept it and payment is easily facilitated on both ends. This tested various aspects of the operations: if the document structs are correct and if the access defined in the contract is working as per requirements. At last, test 3 tested the three types of financing techniques available in the contract. It checked the correctness of the balances of the respective players involved in the financing operations and their access to the information to document structs and the finance structs. It was also checked that suppliers, buyers, banks and investors get their money, tokens and information perfectly. Figure 5.6 shows the test results.

Chapter 6: Future Work and Conclusion

6.1 Future Work

The proof-of-concept demonstrates how supply chain finance can be facilitated and improved by the blockchain. Despite this, it still demonstrates a very basic model that is still far from a full-fledged scalable and deployable solution. In this section, we will discuss the future work from the lenses of operations, security and scalability to make the solution more robust.

The process starts by onboarding the players and current solution assumes that an oracle exists or will exist which will keep a trusted mapping of a business's public address and its business number. In the contract logic, only the owner (or *msg.sender*) can register a company with that business number. Such an oracle solution doesn't exist yet. It is the highest priority to test the feasibility of the development of such an oracle solution. Instead of making every bank performing its own KYC, an oracle will make the KYC easier for the banks because the verified details of companies will be available to retrieve from the blockchain instantly. This will boost SCF adoption by decreasing KYC costs and time since KYC is one of the biggest barriers in supply chain finance adoption (Pasadilla, 2014). For example, when the trade is happening across the border, it becomes extremely expensive in terms of money and time for banks to perform KYC of a buyer in a different country.

In another less disruptive solution, we propose to use a combination of Quorum network and a third trusted party which will govern the onboarding process. Instead of deploying the contract on a public Ethereum blockchain, instance of the smart contract will be deployed on Quorum. On the client-side, the application will have the functionality so that client will be asked to upload the

required documents to prove their legitimacy. The document's authenticity can either be directly scrutinized by a representative from the third party or can be automatically checked by an AI-algorithm in the back end or a combination of both. Once receiving confirmation of authenticity, the company's public address will be whitelisted, and it will receive a unique credential which it can use to onboard the private quorum network and prove authenticity. Inside the network, the business and financing practices can happen fluidly with the logic discussed in the previous Chapter 5. Another possible but ambitious alternative is using Hyperledger Indy + Aries to produce credentials for company/bank ownership and integrating that with the smart contract. This could also be achieved by integrating an authentication layer using verifiable credentials. It must be noted that investors will not be required to undergo this procedure. They will be able to directly interact with the smart contract to buy the issued securities. Permission will only be required to register a company or a financial institution.

A major concern with the current proof-of-concept is the gas cost. It must be optimized to reduce the gas usage before it will be deployed in a public network. If private fork networks like Quorum are used, forked out of the Ethereum Blockchain, the gas usage in them is zero because different consensus mechanisms like Istanbul-BFT are used. But still, the developer must keep an eye on "gas curse" in the Quorum where gas limit must be higher than the transaction usage or transaction will be indefinitely queued (Rocha, 2018). Using Quorum may help to reduce the gas requirements in transactions like "*LiquidateReceivables*" where in the current procedure, the owner must pay a gas cost to withdraw ethers into his/her account. Reducing gas usage will also make the solution significantly more scalable. There are several options to achieve this which must be thoroughly researched: from making changes in the code and logic of smart contract to using side chains,

following upcoming changes in the Ethereum 2.0 and tracking innovative solutions like Wibson which offers to reduce 99% gas costs.

Another notable update in the field of supply chain finance and blockchain is the price rally and market cap increase of DeFi services by a billion-dollars in a span of a month, which aims to completely restructure the banking and financing system (Schroeder, 2020). It is extremely important in future to keep track of such updates and their implications on scalability, security and design of a crypto-finance system.

In the current system, the company is required to input the inventory of the product in its respective units manually. This process is inefficient because it assumes the supplier to be trustworthy of its inventory. An accurate count of inventory is not only required for the smooth working of the application, but it can also be helpful for banks to finance the businesses. Inventory signaling can prove more efficient than cash signaling to depict a firm's operational capabilities (Chod et al., 2019). Moreover, unless the inventory number in the model is trustworthy, SCF instruments like inventory discounting or inventory securitization can't be used. For example, in the case of invoice finance, instead of directly requesting "cash" from the bank, the buyer will be signaling to source the inventory. Researchers also argued that inventory signaling is only feasible when the inventory level can be validated at a low cost.

Instead of relying on the supplier, a hardware oracle can be used which will upload the state of inventory in real-time. The packaged goods can be counted via scanning the bar codes/QR codes automatically. The QR/bar codes can be visible or invisible depending upon the security requirement. Using Invisible QR codes can drastically help to prevent the counterfeiting problem

which is a big challenge in industries like the pharmaceutical industry (Gao et al., 2015). In the case of bulk goods, radar signals can be used to detect the level of the tankers or containers where goods are kept. A more elaborate description is described in Azhos chain which shows that using container's geometry and time required by radar echoes to travel back, the bulk good's level can be precisely calculated (Rudolf et al., n.d.).

In future, a major focus will be on improving the security and confidentiality of transactions. Currently, anyone can find out the public address of the company and query the details about the transaction from that address from the blockchain. For example, suppose public address XA makes transactions of 1 ether to contract address XB. This transaction can be queried and using analytics, patterns can be derived out of grouping all similar transactions to find details about purchases or payments. An alternative considered in future will be the use of Hyperledger Fabric or using Quorum blockchain with added layers of securities. Another major function to be added is before making the purchase request, there must be an encrypted channel for negotiating the price and deadline which after finalization must be automatically converted into a purchase order.

In the current system of shipping, only a tracking ID is generated to find information about the shipping state. But if a smart contract is used at its full capacity, the shipping model will be drastically changed. The shipping carrier will be paid automatically on the correct and timely delivery of goods and will be penalized otherwise. The negotiation channel will be used to decide Incoterms and they will be automatically enforced in the shipment and will be mentioned on the Bill of Ladings.

The current contract does not make use of the exceptional capability of blockchain to track provenance. Once a tracking ID is generated, a new smart contract will be used in future to use it for tracking and tracing purpose. It will keep track of every step of the logistic lifecycle from production to warehousing to distribution and subsequently delivery. This will open a completely new domain of application which can allow multiple use cases based on tracking ID generated at the time of shipping. The children contract will just have to call the main contract with the tracking ID and the subsequent functions will be using it.

The current system of payment is very inefficient. The current process is two steps where the buyer makes the payment to the contract and the entity who is entitled to the receivables burns the REC tokens to withdraw the ether into its account. This must be changed to a single step with added penalties to the buyer in case of defaulting invoice payment. Once the buyer receives the invoice, a smart contract will enforce the payment at the deadline or late payment plus penalty after the deadline. Moreover, in future, every buyer and seller will be given points on a “scale” which denotes the ease of business and reputation with that company. Using all these features, a comprehensive solution will be developed to facilitate negotiations and payment.

Another major concern with the current payment system is the price fluctuations of Ether. Eventually, the actors will have to withdraw their money from a crypto exchange in dollars which will be governed by the price of Ether at that point. To tackle this, stable coins can be used in future which are pegged to fiat currencies or gold. There are several stable coins like Tether, USD coin which maintain 1:1 relationship with USD. In future, a decision will be required from a strategic and technical perspective to choose the appropriate stable coin or whether to completely disregard it and come up with an original solution.

There is tremendous scope to improve the current financing operations. The smart contract can make purchase order finance more fluid, automatic and secure. In the current system, once the purchase order financing is approved by the bank, all the trust of the operation assumes that supplier will manufacture the products, accepts the order again and ships it. In a disruptive future model, using a combination of hardware and software oracle, once the purchase order is approved, it will trigger the manufacturing automatically using a hardware oracle, accept the order and ship it to the supplier instead of involving the supplier at all to prevent fraud.

In the current proof-of-concept, the buyer picks the invoice ID which it wants to get financed and the bank which will finance it. But it is more efficient to first classify the suppliers based on their disruption risk profile, capital level and historical business(Huang, n.d.). In the future, machine learning and AI-based techniques can aid buyers to categorize the suppliers and suggest specific financing techniques for their business using supplier's history and reputation points. In future, there is also an opportunity to make use of bidding to provide the best invoice discount and PO discount to the supplier. In the current contract, the discount rates are hardcoded, but in future, banks will be bidding for different purchase order and invoice finances and this will lead to a whole new dynamics of game theory between suppliers and banks. Moreover, the current contract only allows for complete financing. In future, suppliers and other actors will be able to apply for partial financing of their invoices or orders in the different logistic stage of the order.

The reverse securitization process in the smart contract follows single buyer-led multi-seller securitization, but in future, multi-buyer securitization and supplier-led securitizations will be explored. Fintech platforms found it easier to facilitate PO finance or invoice finance rather than to set up multi-buyer securitization (Hofmann et al., 2017). But it was before the inception of

blockchain and hence multi-buyer securitization was only available on a few large bank driven programs (Miller, 2007). There were several issues to pool risk - it requires at least 20 buyers, and to sustain the ecosystem of buyer and sellers all the invoices accepting finance must be paid early (Hofmann et al., 2017). The changes in future will be at a granular level – buyer/seller will be able to choose and pick the invoice that it wants to securitize and after properly secured communication with another party, the securitization will be initiated.

On the investors' side, strategic decisions will be needed to mitigate the risks – generation of investor tranches and onboarding external support like bilateral guarantees (Nassr & Wehinger, 2015). In the current system, the insurance company only rates the SPV, but in future, the insurance company will play a major role in securing the risk. It will provide a credit increase to Asset Based securities (ABS) by financial guarantees depending upon the rating it provides.

In case the securitization is bank-driven, a strategic choice will have to be made since banks also act as an insurer in most cases. In addition to this, thorough research will be done from a legal perspective to not leave any loophole from the legal end. For example, the current system makes the SPV with a public address as the contract address. The legal implication of this is yet to be studied and if it challenges the law then necessary changes will be needed to be done in future.

The current payment system requires suppliers to withdraw the money from SPV's address on first-come-first-serve basis once the money is deposited by investors. This process needs a reevaluation, and probably a re-design, where supplier will be categorized based on a disclosed criterion which will queue them to withdraw their money from the system.

Lastly, provisions will have to be made in the future which will enable upgrading of the smart contracts. The changes can be stimulated internally or externally. External changes can be brought by upgrading the underlying technologies. Smart contracts, like any other tool, are undergoing constant research and have new features every few months. For example, Ethereum recently launched Solidity compile version 6 and Ethereum 2.0 is imminent. It will need to make changes in the code to reap the full benefits of new upcoming features. It is not possible to upgrade contracts in the current structure since they are directly inherited. To facilitate upgrading, the first change will be dividing smart contract into a contract each for onboarding and different financing instrument and making use of contract abstraction by deploying instances of each.

The interplay of these instances will guarantee current features with access control. There will be data/storage contracts, and a proxy contract and logic contracts inheriting from the data/storage contract.

Data contracts will be permanent and logic contracts will be upgradable by using proxy contract's fall back function to make a delegated call to logic contracts. This will allow logic contracts to make changes to the storage of eternal proxy contracts. Not only this will inhibit a decrease in the gas requirement, but it will also allow better scalability (Gupta, 2018). Figure 6.1 provides a picture for a potential future smart contract which can be used to perform purchase order financing. All the contracts can be imagined in “proxy, logic and storage” combination. The system will be composed of three contracts onboarding, PO-Finance and payment. There will be a constant interplay of functional calls between contracts, a contract and oracle service (hardware and software), and contract and the quorum blockchain in the steps as shown in figure 6.1.

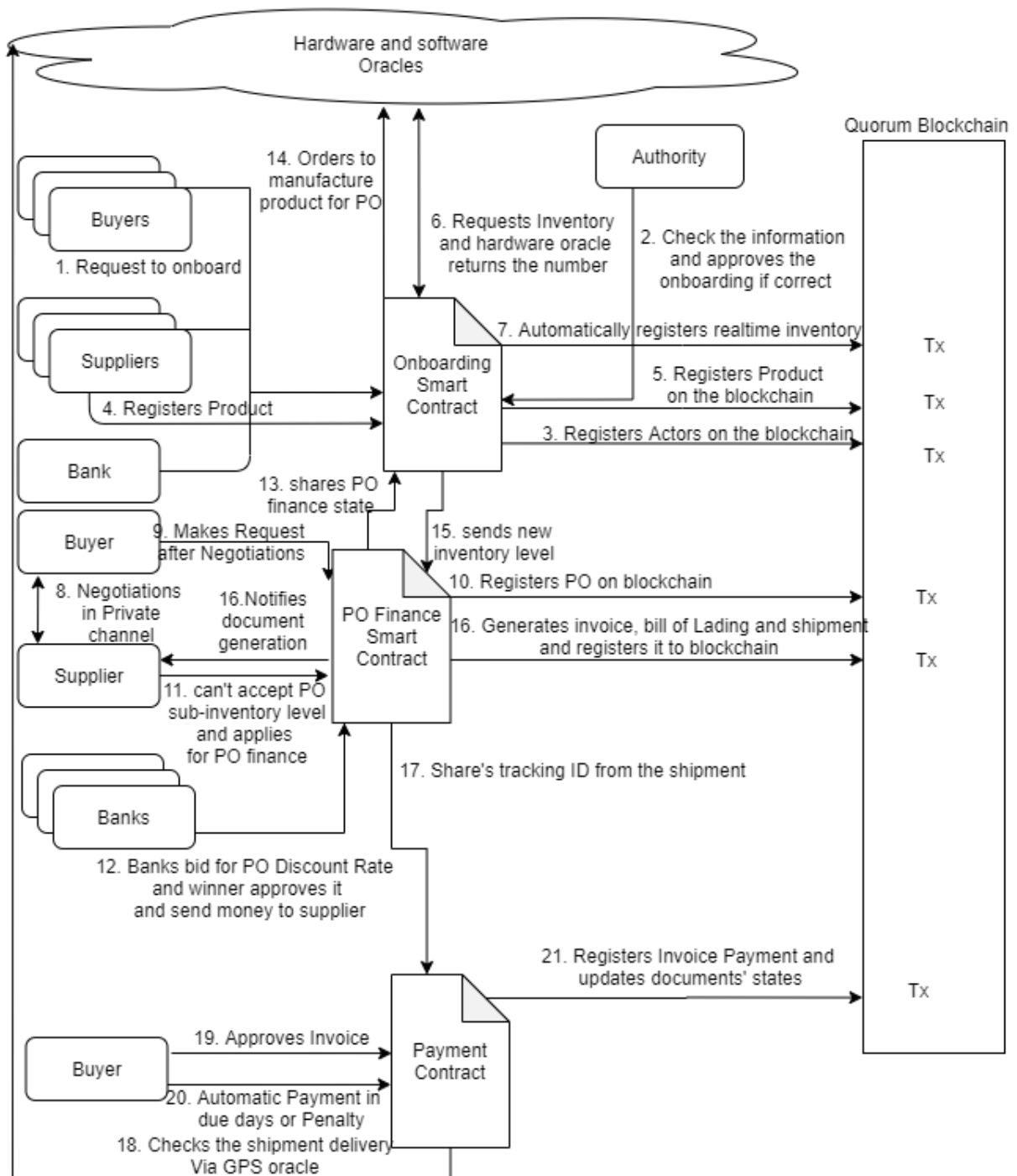


Figure 6.1 Future Model for Blockchain based Purchase Order Financing (constructed by the author)

6.2 Conclusion

In this study, a comprehensive explanation of the various supply chain finance techniques, the differences between them and their importance was laid out. The study then delves deeply into the emerging blockchain technology. In addition to describing the history and inception of blockchain, a technically elaborate description of the technology was made. This included a discussion on types of blockchains, its various components, backend mechanisms and a description of how all the components and algorithms orchestrate together to form a decentralized distributed ledger system. Later, the advantages and disadvantages were listed which can help in the making of proof-of-concept smart contract. Following this, one of the most important elements of blockchains that enable decentralized operations was elaborately described – smart contracts. From its history to its operations, its advantages, limitations and use cases in supply chain management were mentioned which lead to narrow down the potential blockchains for building a proof-of-concept.

As a result of the literature study, a proof-of-concept was made which enabled actors onboarding, normal business operations, supplier-led purchase order financing, reverse factoring and reverse securitization. Ethereum based Solidity was used to create the smart contract and its operations in four different use cases demonstrating how KYC and final settlement can be done using few clicks in a few minutes as compared to the current system in place, which takes considerably more time. Later, it was also depicted how invoice and bill of Ladings fraud can be eliminated with a full-ledged system in place. To demonstrate this, a combination of Truffle suite and Chai.js was used for the 3 unit-tests with 10 subtests in JavaScript.

Blockchain trilemma, a term coined by Ethereum cofounder Vitalik Buterin, is a well-known phenomenon which describes that there exists a constant trade-off between decentralization, scalability and security of a blockchain system. The proof-of-concept is no exception to this and is heavily focused on security and decentralization at the cost of scalability. The design of any blockchain system must consider the desired goal that the system wants to achieve and prioritize the elements from the trilemma which can help achieve it. The Blockchain technology is extremely mercurial and there is a lot of turbulence as a result of constant inventions and research work. It is analogous from the current day internet which started as a science project to share research papers, transformed into a technologically intricate network governing social and economic lives of people with e-commerce and social media. The kind of companies that have spawned as a ramification of this were never envisaged, even by the pundits back then. Blockchain is currently at stage three and four of the hype-cycle: innovation, a peak of inflated expectation, a trough of disillusionment and slope of enlightenment, where people are exploring the true potential of blockchain before the growth plateaus.

At the end of this study, the future directions were laid down which will lead to improvements in the system in terms of scalability and security at the cost of decentralization. To conclude, the proof-of-concept although functional, is at a very preliminary stage and with constant refinement can lead to a holistic supply chain finance solution in the future to improve the current trading operations and benefit the SME in developing and developed countries.

Bibliography

A Short History of #Ethereum. (2019, May 13). ConsenSys.

<https://consensys.net/blog/blockchain-explained/a-short-history-of-ethereum/>

Alharby, M., & Moorsel, A. van. (2017). Blockchain Based Smart Contracts: A Systematic Mapping Study. *Computer Science & Information Technology (CS & IT)*, 125–140.

<https://doi.org/10.5121/csit.2017.71011>

Almasoud, A. S., Eljazzar, M. M., & Hussain, F. (2018). Toward a Self-Learned Smart Contracts. *2018 IEEE 15th International Conference on E-Business Engineering (ICEBE)*, 269–273. <https://doi.org/10.1109/ICEBE.2018.00051>

Alora, A., & Barua, M. K. (2019). Barrier analysis of supply chain finance adoption in manufacturing companies. *Benchmarking: An International Journal*, 26(7), 2122–2145.

<https://doi.org/10.1108/BIJ-08-2018-0232>

Art. 17 GDPR – Right to erasure (‘right to be forgotten’). (2016, April 27). *General Data Protection Regulation (GDPR)*. <https://gdpr-info.eu/art-17-gdpr/>

Berez, S., & Sheth, A. (2007, November 1). Break the Paper Jam in B2B Payments. *Harvard Business Review*, November 2007. <https://hbr.org/2007/11/break-the-paper-jam-in-b2b-payments>

Bitcoin Price Index—Real-time Bitcoin Price Charts. (n.d.). CoinDesk. Retrieved January 18, 2020, from <https://www.coindesk.com/price/bitcoin>

Blockchain Use Cases For Banks In 2020. (2019, September 10). Finextra Research.

<https://www.finextra.com/blogposting/17857/blockchain-use-cases-for-banks-in-2020>

- BTC STUDIOS. (2017, October). *Yes, Bitcoin Can Do Smart Contracts and Particl Demonstrates How*. <https://bitcoinmagazine.com/articles/yes-bitcoin-can-do-smart-contracts-and-particl-demonstrates-how>
- Buterin, V. (n.d.). *A NEXT GENERATION SMART CONTRACT & DECENTRALIZED APPLICATION PLATFORM*. 36.
- Buterin, V. (2015). *Whitepaper | Ethereum Builder's Guide*. <https://ethereumbuilders.gitbooks.io/guide/content/en/whitepaper.html>
- CCN. (2020, January 14). *Where Bitcoin Goes Next After This Week's Stunning Rally*. CCN.Com. <https://www.ccn.com/where-bitcoin-goes-next-after-this-weeks-stunning-rally/>
- Celiz, R. C., De La Cruz, Y. E., & Sanchez, D. M. (2018). Cloud Model for Purchase Management in Health Sector of Peru based on IoT and Blockchain. *2018 IEEE 9th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, 328–334. <https://doi.org/10.1109/IEMCON.2018.8615063>
- Chandrashekhar, G. (2014, June). *Copper financing scandal in China rocks world market*. @businessline. <https://www.thehindubusinessline.com/opinion/columns/g-chandrashekhar/Copper-financing-scandal-in-China-rocks-world-market/article20798365.ece>
- Chod, J., Trichakis, N., Tsoukalas, G., Aspegren, H., & Weber, M. (2019). *On the Financing Benefits of Supply Chain Transparency and Blockchain Adoption*. 67.
- Chohan, U. W. (2017). *Cryptoanarchism and Cryptocurrencies*. *SSRN Electronic Journal*. <https://doi.org/10.2139/ssrn.3079241>

- Devalkar, S. K., & Krishnan, H. (2019). The Impact of Working Capital Financing Costs on the Efficiency of Trade Credit. *Production and Operations Management*, 28(4), 878–889.
<https://doi.org/10.1111/poms.12954>
- Dhillon, V., Metcalf, D., & Hooper, M. (2017). The DAO Hacked. In V. Dhillon, D. Metcalf, & M. Hooper (Eds.), *Blockchain Enabled Applications: Understand the Blockchain Ecosystem and How to Make it Work for You* (pp. 67–78). Apress.
https://doi.org/10.1007/978-1-4842-3081-7_6
- Di Gregorio, M. (2017). *Blockchain: A new tool to cut costs*. PwC.
<https://www.pwc.com/m1/en/media-centre/articles/blockchain-new-tool-to-cut-costs.html>
- Dika, A. (2017). *Ethereum Smart Contracts: Security Vulnerabilities and Security Tools*. 97.
- EDC Canada. (2017). *Competitive-Payment-Terms.pdf*.
<https://creditinstitute.org/UploadedFiles/Newsletter/Competitive-Payment-Terms.pdf>
- Ernst and Young LLP. (2019). *Total cost of ownership for blockchain solutions*. 16.
- Eschenburg. (2015, November 19). *64% of Small Businesses Wait on Late Payments (An Infographic)* [Fundbox]. <https://fundbox.com/blog/late-payments/>
- Fabozzi, F. J., Davis, H. A., & Choudhry, M. (2006). *Introduction to structured finance*. John Wiley.
- Financial Times. (2012). *UK banks hit by record \$2.6bn US fines*.
<https://www.ft.com/content/643a6c06-42f0-11e2-aa8f-00144feabdc0>.
- FINASTRA. (2019, November 18). *Fighting financial crime in Trade and Supply Chain Finance*. Finastra. <https://www.finastra.com/viewpoints/blog/fighting-financial-crime-trade-and-supply-chain-finance>

Fischer, J., & Lynch, A. (1985). *Impossibility of Distributed Consensus with One Faulty Process*.

9.

Floyd, D. (2018, June 10). *The Top 5 Ethereum Dapps By Daily Active Users*. CoinDesk.

<https://www.coindesk.com/top-5-ethereum-dapps-daily-active-users>

Gao, Z., Zhai, G., & Hu, C. (2015, October 1). *The Invisible QR Code*.

<https://doi.org/10.1145/2733373.2806398>

Gatteschi, V., Lamberti, F., Demartini, C., Pranteda, C., & Santamaría, V. (2018). To Blockchain or Not to Blockchain: That Is the Question. *IT Professional*, 20(2), 62–74.

<https://doi.org/10.1109/MITP.2018.021921652>

Gencer, A. E., Basu, S., Eyal, I., van Renesse, R., & Sirer, E. G. (2018). Decentralization in Bitcoin and Ethereum Networks. *ArXiv:1801.03998 [Cs]*.

<http://arxiv.org/abs/1801.03998>

Glaser, F. (2017). *Pervasive Decentralisation of Digital Infrastructures: A Framework for Blockchain enabled System and Use Case Analysis*. 10.

Global Supply Chain Finance Forum. (2015). *ICC-Standard-Definitions-for-Techniques-of-Supply-Chain-Finance-Global-SCF-Forum-2016.pdf*.

<http://supplychainfinanceforum.org/ICC-Standard-Definitions-for-Techniques-of-Supply-Chain-Finance-Global-SCF-Forum-2016.pdf>

Goldman Sachs. (2016). *Blockchain: Putting theory in practice*.

<https://www.coursehero.com/file/16984495/Goldman-Sachs-report-Blockchain/>

Greenspan, G. (2016, April 18). *Why Many Smart Contract Use Cases Are Simply Impossible—*

CoinDesk. CoinDesk. <https://www.coindesk.com/three-smart-contract-misconceptions>

- Gupta, M. (2018, September 6). *How to make smart contracts upgradable!* / *Hacker Noon*.
Hackernoon. <https://hackernoon.com/how-to-make-smart-contracts-upgradable-2612e771d5a2>
- Gustin, D. (2018, June 14). *KYC for Supply Chain Finance is an Unmitigated Disaster*.
<https://spendmatters.com/tfmatters/kyc-for-supply-chain-finance-is-an-unmitigated-disaster/>
- Hofmann, E., Strewe, U. M., & Bosia, N. (2017). *Supply Chain Finance and Blockchain Technology: The Case of Reverse Securitisation*. Springer.
- Huang, X. (n.d.). *Financing Disruptive Suppliers with Smart Contracts*. 35.
- Hyperledger_DataSheet_11.18_Digital.pdf*. (n.d.). Retrieved June 24, 2020, from
https://www.hyperledger.org/wp-content/uploads/2018/11/Hyperledger_DataSheet_11.18_Digital.pdf
- ISO/FDIS 22739 Blockchain and distributed ledger technologies—Vocabulary*. (2020).
- Josie Cox. (2020, June 28). *Debate continues over value of cryptocurrencies in COVID-19 market*. Raconteur. <https://www.raconteur.net/finance/trading-strategies-2020/covid-crypto-value>
- Kadam, S. K. (2018). *Review of Distributed Ledgers: The technological Advances behind cryptocurrency*. 6.
- Kaligotla, C., & Macal, C. M. (2018). A GENERALIZED AGENT BASED FRAMEWORK FOR MODELING A BLOCKCHAIN SYSTEM. *2018 Winter Simulation Conference (WSC)*, 1001–1012. <https://doi.org/10.1109/WSC.2018.8632374>

- Karaffa, R. (2012, March 23). *SCM30: What Can We Learn From Supply Chain Management Mistakes?* Kinaxis. <https://www.kinaxis.com/en/blog/scm30-what-can-we-learn-from-supply-chain-management-mistakes>
- Lamoureaux, J.-F., & Evans, T. A. (2011). Supply Chain Finance: A New Means to Support the Competitiveness and Resilience of Global Value Chains. *SSRN Electronic Journal*. <https://doi.org/10.2139/ssrn.2179944>
- Lamport, L., Shostak, R., & Pease, M. (1982). The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3), 20.
- Lekkakos, S., & Serrano, A. (2016). Supply chain finance for small and medium sized enterprises: The case of reverse factoring. *International Journal of Physical Distribution & Logistics Management*, 46, 367–392. <https://doi.org/10.1108/IJPDLM-07-2014-0165>
- Mark B. Solomon. (2018, January 16). *Maersk, IBM launch first blockchain joint venture for trade, transportation*. DC Velocity. <https://www.dcvelocity.com/articles/29429-maersk-ibm-launch-first-blockchain-joint-venture-for-trade-transportation>
- Marr, B. (2019a, January 7). *5 Technology Trends That Will Make Or Break Many Careers In 2019*. Forbes. <https://www.forbes.com/sites/bernardmarr/2019/01/07/5-technology-trends-that-will-make-or-break-many-careers-in-2019/>
- Marr, B. (2019b, September 3). *The 7 Biggest Technology Trends In 2020 Everyone Must Get Ready For Now*. Forbes. <https://www.forbes.com/sites/bernardmarr/2019/09/30/the-7-biggest-technology-trends-in-2020-everyone-must-get-ready-for-now/>
- Maymounkov, P., & Mazières, D. (2002). Kademlia: A peer-to-peer information system based on the xor metric. *Lecture Notes in Computer Science / SpringerLink*, 2429, pages 53–65.

- Miller, A. (2007, September 15). *Trade Services—Pooled Payables Securitisation*. Global Trade Review (GTR). <https://www.gtreview.com/news/global/trade-services-pooled-payables-securitisation/>
- Mingxiao, D., Xiaofeng, M., Zhe, Z., Xiangwei, W., & Qijun, C. (2017). A review on consensus algorithm of blockchain. *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, 2567–2572. <https://doi.org/10.1109/SMC.2017.8123011>
- Nassr, I. K., & Wehinger, G. (2015). Unlocking SME finance through market-based debt: Securitisation, private placements and bonds. *OECD Journal: Financial Market Trends*, 2014(2), 89–190. <https://doi-org.ezproxy.library.ubc.ca/10.1787/fmt-2014-5js3bg1g53ln>
- Noah, D. (2020, May 4). *8 Documents Required for International Shipping*. <https://www.shippingsolutions.com/blog/documents-required-for-international-shipping>
- Pasadilla, G. O. (2014). *Regulatory Issues Affecting Trade and Supply Chain Finance*. APEC Policy Support Unit.
- Petersson, D. (2018, October 24). *How Smart Contracts Started And Where They Are Heading*. Forbes. <https://www.forbes.com/sites/davidpetersson/2018/10/24/how-smart-contracts-started-and-where-they-are-heading/>
- Pope, S. (2019, October 16). *Blockchain To Be A Gamechanger For Global Shipping*. Forbes. <https://www.forbes.com/sites/stephenpope/2019/10/16/blockchain-to-be-a-gamechanger-for-global-shipping/>
- PricewaterhouseCoopers. (2017). *Structured payables: Should your trade payables be classified as debt?* PwC. <https://www.pwc.com/us/en/services/audit-assurance/accounting-advisory/structured-payables.html>

- PYMNTS. (2019, April 15). Supply Chain Finance Stumped By KYC. *PYMNTS.Com*.
<https://www.pymnts.com/news/b2b-payments/2019/supply-chain-finance-survey/>
- Ragan, C. (2008). *Filling A Shrinking Gap*.
https://www.mcgill.ca/economics/files/economics/filling_a_shrinking_gap_-_edcs_changing_role_in_the_market_for_export-credit_insurance.pdf
- Remix—Ethereum IDE. (n.d.). Retrieved June 27, 2020, from
<https://remix.ethereum.org/#optimize=false&evmVersion=null&version=soljson-v0.6.6+commit.6c089d02.js>
- Rocha, A. de la. (2018, December 9). *How to survive the gas curse in Quorum?* Medium.
<https://medium.com/coinmonks/how-to-survive-the-gas-curse-in-quorum-7b2c8fea2540>
- Rosa, E., D'Angelo, G., & Ferretti, S. (2019). Agent-Based Simulation of Blockchains. In G. Tan, A. Lehmann, Y. M. Teo, & W. Cai (Eds.), *Methods and Applications for Modeling and Simulation of Complex Systems* (Vol. 1094, pp. 115–126). Springer Singapore.
https://doi.org/10.1007/978-981-15-1078-6_10
- Rudolf, B., Kuhs, M., Baer, M., & Zintl, S. (n.d.). *AZHOS Whitepaper*. 27.
- Schroeder, S. (2020, July 17). *DeFi could become the next big thing in finance*. Mashable.
<https://mashable.com/article/defi-growing-fast/>
- Sheffi, Y. (2015). *The Financial Crisis and the Money Supply Chain – finance – CSCMP's Supply Chain Quarterly*.
<https://www.supplychainquarterly.com/topics/finance/20151223-the-financial-crisis-and-the-money-supply-chain/>

- Sommer, M., & O’Kelly, R. (2017). *SUPPLY CHAIN FINANCE: RIDING THE WAVES*.
<https://www.oliverwyman.com/content/dam/oliverwyman/v2/publications/2017/dec/Supply-chain-finance-Final.pdf>
- Stein, P., Pinar Ardic, O., & Hommes, M. (2013). *Closing the Credit Gap for Formal and Informal Micro, Small, and Medium Enterprises*.
- Szabo, N. (1997). Formalizing and Securing Relationships on Public Networks. *First Monday*, 2(9). <https://doi.org/10.5210/fm.v2i9.548>
- Szabo, N. (2008, December 27). Unenumerated: Bit gold. *Unenumerated*.
<https://unenumerated.blogspot.com/2005/12/bit-gold.html>
- Szalachowski, P. (2018). (Short Paper) Towards More Reliable Bitcoin Timestamps. 2018 *Crypto Valley Conference on Blockchain Technology (CVCBT)*, 101–104.
<https://doi.org/10.1109/CVCBT.2018.00018>
- Trade finance is going open account*. (2018, March 21). Nordea.
<https://insights.nordea.com/en/ideas/trade-finance-is-going-open-account/>
- Vincent, J. (2019, July 4). *Bitcoin consumes more energy than Switzerland, according to new estimate*. The Verge. <https://www.theverge.com/2019/7/4/20682109/bitcoin-energy-consumption-annual-calculation-cambridge-index-cbeci-country-comparison>
- Wang, S., Ouyang, L., Yuan, Y., Ni, X., Han, X., & Wang, F.-Y. (2019). Blockchain-Enabled Smart Contracts: Architecture, Applications, and Future Trends. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 49(11), 2266–2277.
<https://doi.org/10.1109/TSMC.2019.2895123>

Wharf Street Strategies. (2019, May). *Blockchain Use Case: KYC And AML*.

<https://medium.com/wharfstreetstrategies/blockchain-use-case-kyc-and-aml-7357578ffe18>

Wood, D. G. (2015). *ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER*. 39.

Wuttke, D., Rosenzweig, E., & Heese, H. (2019). An empirical analysis of supply chain finance adoption. *Journal of Operations Management*, 65, 242–261.

<https://doi.org/10.1002/joom.1023>

Yannik, Z. (2019, January 22). A clarification on the perpetual discussion of Bitcoin's timestamp [Educational]. *A Clarification on the Perpetual Discussion of Bitcoin's Timestamp*.

<https://hackernoon.com/a-clarification-on-the-perpetual-discussion-of-bitcoins-timestamp-5597859a9193>

Appendices

Appendix A RegisterCompany.sol

```
pragma Solidity >=0.5.0 <0.6.9;
pragma experimental ABIEncoderV2;

contract RegisterCompany {

    address payable admin;

    constructor() public {
        admin = msg.sender;
    }

    event CompanyRegistered( uint companyId, string name);
    event ProductRegistered (uint productId, string prodName);
    event Error(string message);

    struct Company {
        uint companyId; // unique ID of every company
        address payable owner; // address that owns that company-- each address can have multiple
companies
        string companyName; // name of the company
        string location; // location for delivery
        string businessNumber; // a unique identification number to match with public address
        bool isInsuranceCompany; // true if insurance company
    }

    struct Product {
        uint productId; // unique Id of product
        string prodName; // product's name
        uint companyId; // company ID which has the product registered
        uint sellingPrice; // price in wei for the product
    }

    uint private initProductId = 0; // initial Id of the product
    uint private companyId = 0; // initial company ID
    uint internal totalCompanies = 0; // total number of the companies
    uint internal totalProducts = 0; // total number of products

    mapping (uint => Company) public companies; // mapping containing all the companies
(id=>struct)
```

```

mapping (uint => Product) public products; // mapping containing all the products (id=>struct)
mapping (address => uint) private ownedCompanies; // total number of companies owned by
an address

```

```

function generateCompanyId() internal returns(uint){
    companyId +=1;
    return companyId;

}

```

```

function createCompany( string memory _companyName, string memory _location, string
memory _businessNumber) public returns( uint) {
    // check if businessNumber is right using api oracle
    // check if name and address matches the businessNumber
    //require businessNumber =9digits
    require(bytes(_companyName).length >0 && (bytes(_location).length >0));
}

```

```

Company memory company;

```

```

    company = Company(generateCompanyId(),
msg.sender,_companyName,_location,_businessNumber,false);
    uint _companyId = company.companyId;
    companies[_companyId] = company;

    ownedCompanies[msg.sender] +=1;
    totalCompanies +=1;

    emit CompanyRegistered(_companyId, _companyName);
    return _companyId;
}

```

```

function getCompany (uint _companyId) public view returns (
    uint companyId,
    address payable owner,
    string memory companyName,
    string memory location,
    string memory businessNumber,
    bool isInsuranceCompany)
{
    Company memory company = companies[_companyId];

    return(
        company.companyId,

```

```

        company.owner,
        company.companyName,
        company.location,
        company.businessNumber,
        company.isInsuranceCompany
    );
}
function generateProductId()private returns(uint){
    initProductId +=1;
    return initProductId;
}

function registerProduct ( uint _companyId, string memory _prodName,uint _sellingPrice )
public returns(bool success) {
    // oracle if company actually manufactures that _product
    // don't register if already registered
    require( companies[_companyId].owner == msg.sender);
    require(bytes(_prodName).length >0);

    Product memory product;
    product = Product(generateProductId(),_prodName, _companyId,_sellingPrice);
    uint _productId = product.productId;
    products[_productId] = product;
    // productToCompany[_productId] = _companyId;
    // companyOwnedProductNumber[_companyId]+=1;
    totalProducts+=1;
    emit ProductRegistered(_productId,_prodName);
    return true;
}

function getProduct(uint _productId) public view returns (uint productId,
    string memory prodName,
    uint companyId,
    uint sellingPrice
){

    Product memory product = products[_productId];
    return(
        product.productId,
        product.prodName,
        product.companyId,
        product.sellingPrice
    );
}
}

```


Appendix B CompleteFinancingContract.sol

```
pragma Solidity >=0.5.0 <0.6.9;
pragma experimental ABIEncoderV2;

import "./SafeMath.sol";
import "./RegisterCompany.sol";
import "./CompleteFinancingContractInterface.sol";

contract CompleteFinancingContract is CompleteFinancingContractInterface,
RegisterCompany{

    address public contractAddress;// address of the contract
    string public REC = "REC for Recievables"; // name of the REC token
    string public SEC = "SEC for Securities"; // name of SEC token
    using SafeMath for uint256;

    constructor() public {
        contractAddress = address(this);
    }

    struct PurchaseOrderRequest{
        uint purchaseOrderId;// unique ID of the order
        uint productId; // the product ordered
        uint buyerCompanyId; // the company ordering
        uint qty; // quantity ordered
        bool isAccepted; // whether order is accepted
        // put time for PO, invoice,shipping
    }

    struct Invoice{
        uint invoiceId; // unique ID of invoice
        uint orderId; // order for which invoice is associated
        uint paymentdeadline; // the payment deadline to be decided by the seller
        bool isInvoiceApproved; // is invoice Approved by the buyer confirming Shipment
        bool isPaymentDoneByBuyer; // true if payment made by the buyer
        bool isPaymenttakenBySupplier; // true if payment taken by the supplier
        bool isFinanceRequested; // true if finance requested by the buyer
        bool isInvoiceFinanceAccepted; // true if invoice FInance is accepted by the bank
        uint timeOfCreation; // time of creation of invoice for automatic payment in the future
    }

    struct BillOfLading{
        uint bolId; // unique ID for bill of lading
```

```

uint orderId; // order for which BOL is made
uint buyerCompanyId; // company Id which is getting the bill of Lading
uint carrierId; // carrier which will transport
string carrierName; // carrier's name
}

struct Shipment{
    uint trackingId; // tracking ID for the provenance contract
    uint carrierId; // carrier ID
    string deliveryAddress; // address of the buyer
    uint qty; // order quantity
    bool isShipmentRecieved; // true if shipment delivered
}

struct Bank{
    uint bankId; //bank's unique ID
    string bankName; // name of the bank
    string bankCode; // unique code of bank associated with public address
    address payable bankAddress; // bank's public address
}

struct PoFinance{
    uint poFinanceId; // unique purchase order finance id
    uint bankId; // bank which can approve or decline the po
    uint orderId; // the order for which finance has been applied
    uint buyerCompanyId; // the company which made the request
    uint value; // value of the order
    bool isApproved; // true if the finance approved
    uint timeOfApproval; // time at which finance is approved
}

struct InvoiceFinance{
    uint invoiceFinanceId; // unique ID of invoice finance
    uint bankId; // bank's ID which can approve or decline the invoice finance
    uint invoiceId; // the invoice's ID for which finance is applied
    uint buyerCompanyId; // buyer company's ID
    uint value; // value of the invoice
    bool isApproved; // true if finance is Approved
}

struct SPV{
    uint spvId; //unique ID of the SPV
    uint issuerId; // ID of the buyer which issues Security
    uint insuranceCompanyId; // ID of the insurance company to rate

```

```

    uint8 rating; // rating out of 5
}

struct Security{
    uint sId; //unique ID of the Security
    uint amount; // amount of securities issued -- SEC tokens
    uint buyerCompanyId; // the companyID which issues the Security
    uint spvId; // SPV generated for Security
}

struct Note{
    uint noteId; // unique ID of the Note
    uint sId; // Security associated to the note
    uint amount; // amount of SEC tokens
    address payable owner; // owner's public address
    bool isRedeemed; // true if note is redemmed for ether
}

event Faliure(string message);
event BankRegistered(uint bankId, string bankName);
event PoFinanceApproved(uint poFinanceId);
event OrderAccepted(uint _orderId);
event ShipmentDelivered(uint orderId);
event InvoiceFinanceApproved(uint invoiceFinanceId);
event PaymentMade(uint _invoiceId);

mapping(address => mapping(address => uint256)) private allowances; // REC token from
address to address
mapping(address => mapping(address => uint256)) private secAllowances; //SEC tokens from
address to address

mapping (uint => uint) private productToInventory; // inventory associated to the product ID
mapping (uint => PurchaseOrderRequest) private purchaseOrderRequests; // all the
purchaseOrderRequests
mapping (uint => bool) private isPOFinanceRequested; // whether PO finance is Requested
mapping (uint => Invoice) private invoices; // stores all the invoices to their IDs
mapping (uint => BillofLading) private billofLadings; // all bill of ladins
mapping (uint => Shipment) private orderToShipment; // all the shipments indexed by orderID
mapping (uint=> Bank) public banks; // all the banks
mapping (address => uint) private ownedBanks; // number of banks to a public address
mapping (uint => PoFinance) private poFinances; // all the poFinances
mapping (uint => InvoiceFinance) private invoiceFinances; // all the invoiceFinances

```

```

    mapping (uint => uint) private companyOwnedFinanceAcceptedInvoices; // number of
invoices accepting invoiceFinances
    mapping (uint => SPV) public spvs; //all the spvs
    mapping (uint => Security) public securities; //all the securities
    mapping (uint => Note) private notes; // all the notes

    uint private orderId =0; //initial orderId
    uint private totalbolId = 0; //initial bill of Lading ID
    uint private trackingId = 0; //initial shipment ID
    uint private totalInvoices = 0; //initial invoice ID

    uint private bankId =0; //bank's ID
    uint8 public poDiscountRate= 80; // po discount in percentage
    uint8 public invoiceDiscountRate = 90; // invoice discount in percentage

    uint private totalpof = 0; // initial poFinance ID and total poFinances
    uint private totalInvoiceFinances = 0; // initial invoice ID and total invoice finances

    uint private sId = 0; // initial security ID
    uint private spvId = 0; // initial spv ID
    uint private noteId = 0; // initial note ID

    function manufactureProd(uint _productId, uint _amount) public returns (bool success) {
        require(companies[products[_productId].companyId].owner == msg.sender, "you are not the
owner of the product");
        productToInventory[_productId] = productToInventory[_productId].add(_amount);
        return true;
    }

    function getInventory(uint _productId) public view returns (uint){
        require(companies[products[_productId].companyId].owner == msg.sender, "Not your
product");
        return productToInventory[_productId];
    }

    function removeProduct(uint _productId) public returns (bool success){

        uint _companyId = products[_productId].companyId;
        address owner = companies[_companyId].owner;
        require(msg.sender == owner && products[_productId].companyId>0, "You are not the
owner of the Product");

        productToInventory[_productId] =0;
        delete products[_productId];

```

```

        return true;
    }

    function approveReceivables(uint _qty, uint _price) internal pure returns(uint _recievables){
        _recievables = _qty.mul(_price);
        return _recievables;
    }

    function approveRecievables(address _from , address _to, uint256 _value) private returns
    (bool success) {
        allowances[_from][_to] = allowances[_from][_to].add(_value);
        return true;
    }

    function registerBank(string memory _bankName, string memory _bankCode )public returns
    (bool success){
        require(ownedBanks[msg.sender] <1," Cannot register more than one bank per address.");
        bankId = bankId.add(1);
        Bank memory bank;
        bank = Bank(bankId,_bankName, _bankCode, msg.sender);
        banks[bankId] = bank;
        ownedBanks[msg.sender] = ownedBanks[msg.sender].add(1);
        emit BankRegistered(bankId,_bankName);
        return true;
    }

    function getBank(uint _bankId) public view returns (
        uint bankId,
        string memory bankName,
        string memory bankCode,//confirm with prof
        address payable bankAddress){
        Bank memory bank = banks[_bankId];
        return(
            bank.bankId,
            bank.bankName,
            bank.bankCode,
            bank.bankAddress
        );
    }

    function makePurchaseOrderRequest(uint _productId, uint _qty, uint _buyerCompanyId )
    public returns(bool success){
        require(companies[_buyerCompanyId].owner == msg.sender &&
        companies[products[_productId].companyId].owner != msg.sender ,"check the company ID or
        you might be ordering your own product");
    }

```

```

        orderId = orderId.add(1);
        purchaseOrderRequests[orderId] = PurchaseOrderRequest(orderId,_productId,
        _buyerCompanyId, _qty,false);
        isPOFinanceRequested[orderId] = false;
        // emit RequestSent(orderId);
        return true;
    }

function getPurchaseOrderRequest(uint _orderId) public view returns(uint purchaseOrderId,
    uint productId,
    uint buyerCompanyId,
    uint qty,
    bool isAccepted){
    address poBankAddress = banks[poFinances[getpofId(_orderId)].bankId].bankAddress;
    // address invoiceFinanceBankAddress =
banks[invoiceFinances[getinvoiceFinanceId(_orderId)].bankId].bankAddress;
    require(msg.sender ==
companies[products[purchaseOrderRequests[_orderId].productId].companyId].owner ||
msg.sender == companies[purchaseOrderRequests[_orderId].buyerCompanyId].owner ||
msg.sender == poBankAddress
    , "access denied");

    PurchaseOrderRequest memory purchaseOrderRequest =
purchaseOrderRequests[_orderId];
    return (
        purchaseOrderRequest.purchaseOrderId,
        purchaseOrderRequest.productId,
        purchaseOrderRequest.buyerCompanyId,
        purchaseOrderRequest.qty,
        purchaseOrderRequest.isAccepted
    );
}

function acceptOrder( uint _orderId, uint _carrierId, uint _paymentDueDays, bool
_invoiceFinanceAccepted) public returns (bool success){

require(companies[products[purchaseOrderRequests[_orderId].productId].companyId].owner ==
msg.sender && purchaseOrderRequests[_orderId].isAccepted == false, "cannot accept the
order");
    uint _productId = purchaseOrderRequests[_orderId].productId;
    uint _qty = purchaseOrderRequests[_orderId].qty;
    uint _currentInventory = productToInventory[_productId];
    if (_qty <= _currentInventory){

```

```

        productToInventory[purchaseOrderRequests[_orderId].productId] =
productToInventory[purchaseOrderRequests[_orderId].productId].sub(purchaseOrderRequests[_
orderId].qty);
        purchaseOrderRequests[_orderId].isAccepted = true;

        //getter for pof
        if(poFinances[getpofId(_orderId)].isApproved ==true){
            _invoiceFinanceAccepted = false;
        }
        makeInvoice(_orderId,_paymentDueDays,_invoiceFinanceAccepted);
        makeBillOfLading(_orderId,_carrierId);
        shipThisOrder(_carrierId,_orderId);
        emit OrderAccepted(_orderId);
    }
    else {
        emit Faliure("Not enough inventory! Cannot Accpet Order. Manufacture remaining or
Request for PO Finance! ");
    }
    return true;
}

```

```

function makeInvoice( uint _orderId, uint _paymentDueDays, bool
_invoiceFinanceAccepted) private returns (bool success) {

```

```

    bool _isFinanceRequested = false;
    uint _timeofCreation = now;
    totalInvoices = totalInvoices.add(1);
    invoices[totalInvoices] = Invoice(totalInvoices,_orderId, _paymentDueDays,false,
false,false,_isFinanceRequested,_invoiceFinanceAccepted, _timeofCreation);
    if(_invoiceFinanceAccepted == true){

```

```

companyOwnedFinanceAcceptedInvoices[purchaseOrderRequests[_orderId].buyerCompanyId]
=
companyOwnedFinanceAcceptedInvoices[purchaseOrderRequests[_orderId].buyerCompanyId].
add(1);
    }
    return true;
}

```

```

function makeBillOfLading(uint _orderId, uint _carrierId) private returns (bool success) {
    totalbolId = totalbolId.add(1);
    uint _buyerCompanyId = purchaseOrderRequests[_orderId].buyerCompanyId;
    string memory _carrierName = companies[_carrierId].companyName;

```

```

        billOfLadings[totalbolId] =
BillOfLading(totalbolId,_orderId,_buyerCompanyId,_carrierId,_carrierName);
        return true;
    }

    // decline Request function is not needed. Buyer can just not accept a request

    function shipThisOrder( uint _carrierId, uint _orderId) private returns (bool success) {
        require(purchaseOrderRequests[_orderId].isAccepted = true);
        // vehicleNo = vehicleNo.add(1);
        trackingId = trackingId.add(1);
        uint _qty = purchaseOrderRequests[_orderId].qty;// determine by for loop
        // uint _productId = purchaseOrderRequests[_orderId].productId; // determine by QR and
hash
        uint _bolId = getBillOfLadingID(_orderId);
        string memory _receiverLocation =
companies[purchaseOrderRequests[_orderId].buyerCompanyId].location; // check from the
location GPS
        // assuming right checkQuantity and ProductId
        string memory _deliveryAddress =
companies[billOfLadings[_bolId].buyerCompanyId].location;

        orderToShipment[_orderId] = Shipment(trackingId,_carrierId,_deliveryAddress, _qty,false
);
        return true;
    }

    function getInvoiceId( uint _orderId) public view returns( uint invoiceId){

        address poBankAddress = banks[poFinances[getpofId(_orderId)].bankId].bankAddress;
        address invoiceFinanceBankAddress =
banks[invoiceFinances[getinvoiceFinanceId(_orderId)].bankId].bankAddress;
        require(msg.sender ==
companies[products[purchaseOrderRequests[_orderId].productId].companyId].owner ||
msg.sender == companies[purchaseOrderRequests[_orderId].buyerCompanyId].owner
        || msg.sender == poBankAddress || msg.sender == invoiceFinanceBankAddress,"Access
denied for this invoice.");

        for (uint i = 0; i<= totalInvoices; i++){
            if( invoices[i].orderId == _orderId){
                return i;
            }
        }
    }
}

```



```

function getInvoice(uint _orderId) public view returns(
    uint invoiceId,
    uint orderId,
    uint paymentdeadline,
    bool isInvoiceApproved,
    bool isPaymentDoneByBuyer,
    bool isPaymenttakenBySupplier,
    bool isFinanceRequested,
    bool isInvoiceFinanceAccepted,
    uint timeOfCreation
){

    Invoice memory invoice = invoices[getInvoiceId(_orderId)];
    return( invoice.invoiceId,
            invoice.orderId,
            invoice.paymentdeadline,
            invoice.isInvoiceApproved,
            invoice.isPaymentDoneByBuyer,
            invoice.isPaymenttakenBySupplier,
            invoice.isFinanceRequested,
            invoice.isInvoiceFinanceAccepted,
            invoice.timeOfCreation
        );
}

```

```

function getBillofLadingID( uint _orderId) private view returns(uint bolId){

    require(msg.sender ==
companies[products[purchaseOrderRequests[_orderId].productId].companyId].owner ||
msg.sender == companies[purchaseOrderRequests[_orderId].buyerCompanyId].owner,"access
denied for this Bill of Lading");

    for (uint i = 0; i<= totalbolId; i++){
        if( billofLadings[i].orderId == _orderId){
            return i;
        }
    }
}

```

```

function getBillofLading( uint _orderId) public view returns(
    uint bolId,
    uint orderId,
    uint buyerCompanyId,

```

```

uint carrierId,
string memory carrierName
){
    BillOfLading memory billOfLading = billofLadings[getBillofLadingID(_orderId)];
    return(
        billOfLading.bolId,
        billOfLading.orderId,
        billOfLading.buyerCompanyId,
        billOfLading.carrierId,
        billOfLading.carrierName
    );
}

function getShipment( uint _orderId) public view returns(
    uint trackingId,
    uint carrierId,
    string memory deliveryAddress,
    uint qty,
    bool isShipmentRecieved
){
    address poBankAddress = banks[poFinances[getpofId(_orderId)].bankId].bankAddress;
    address invoiceFinanceBankAddress =
banks[invoiceFinances[getinvoiceFinanceId(_orderId)].bankId].bankAddress;
    require(msg.sender ==
companies[products[purchaseOrderRequests[_orderId].productId].companyId].owner ||
msg.sender == companies[purchaseOrderRequests[_orderId].buyerCompanyId].owner
    || msg.sender == poBankAddress || msg.sender == invoiceFinanceBankAddress,"Access
denied for this shipment.");
    Shipment memory shipment = orderToShipment[_orderId];
    return(
        shipment.trackingId,
        shipment.carrierId,
        shipment.deliveryAddress,
        shipment.qty,
        shipment.isShipmentRecieved
    );
}

function confirmShipment(uint _invoiceId) public returns (bool success){
    // use oracle to confirm the location
    // notofy seller of delivery if location has reached
    // buyer has to checkQuantity
    require(orderToShipment[invoices[_invoiceId].orderId].isShipmentRecieved == false &&
invoices[_invoiceId].isInvoiceApproved == false,"shipment already delivered.");
}

```

```

        address payable _reciever =
companies[purchaseOrderRequests[invoices[_invoiceId].orderId].buyerCompanyId].owner;
        require(msg.sender == _reciever,"You are not the reciever!");
        address payable _seller =
companies[products[purchaseOrderRequests[invoices[_invoiceId].orderId].productId].companyId].owner;
        //assuming correct qty
        uint _qty = purchaseOrderRequests[invoices[_invoiceId].orderId].qty;
        uint _price =
products[purchaseOrderRequests[invoices[_invoiceId].orderId].productId].sellingPrice;
        invoices[_invoiceId].isInvoiceApproved = true;
        orderToShipment[invoices[_invoiceId].orderId].isShipmentRecieved = true;
        uint _value = approveReceivables(_qty,_price);
        // if qty wrong then notify Bank and paymen
        if(invoices[_invoiceId].isFinanceRequested == true){
            // Invoice Financing --do nothing.
        }
        else if(isPOFinanceRequested[invoices[_invoiceId].orderId] == false){
            approveRecievables(msg.sender,_seller, _value);
        }
        else if(isPOFinanceRequested[invoices[_invoiceId].orderId] == true){
            // Purchase Order Financing
            uint _poFinanceId = getpofId(invoices[_invoiceId].orderId);
            if(poFinances[_poFinanceId].isApproved==true){
                address payable _bankAddress =
banks[poFinances[_poFinanceId].bankId].bankAddress;
                approveRecievables(msg.sender, _bankAddress,_value);
                invoices[_invoiceId].isPaymenttakenBySupplier =true;
            }
            else{
                emit Faliure("finance not approved but shipment reached!");
            }
        }
        emit ShipmentDelivered(invoices[_invoiceId].orderId);
        return true;
    }

function cancelOrder(uint _orderId) public returns(bool success) {
    uint _poFinanceId = getpofId(_orderId);
    require(purchaseOrderRequests[_orderId].isAccepted == false &&
companies[purchaseOrderRequests[_orderId].buyerCompanyId].owner == msg.sender &&
poFinances[_poFinanceId].isApproved == false,"cannot cancel this order.");
    delete purchaseOrderRequests[_orderId];
    return true;
}

```

```

function makePayment( uint _invoiceId) public payable returns (bool success) {
    // _buyer use this to send money to contract
    require(msg.sender ==
companies[purchaseOrderRequests[invoices[_invoiceId].orderId].buyerCompanyId].owner &&
invoices[_invoiceId].isPaymentDoneByBuyer == false &&
invoices[_invoiceId].isInvoiceApproved == true,"check the invoice ID or the amount entered.");
    uint _qty = orderToShipment[invoices[_invoiceId].orderId].qty;
    uint _price =
products[purchaseOrderRequests[invoices[_invoiceId].orderId].productId].sellingPrice;
    uint _value = approveReceivables(_qty,_price);
    require(msg.value >= _value);
    if(msg.value > _value ) {
        address(uint160(msg.sender)).transfer(msg.value.sub(_value));
    }
    invoices[_invoiceId].isPaymentDoneByBuyer = true;
    emit PaymentMade(_invoiceId);
    return true;
}

```

```

function LiquidateReceivables( uint _invoiceId) public returns (bool success){

    address payee =
companies[purchaseOrderRequests[invoices[_invoiceId].orderId].buyerCompanyId].owner;

    address seller =
companies[products[purchaseOrderRequests[invoices[_invoiceId].orderId].productId].companyId].owner;

    uint _amount  = allowances[payee][msg.sender];
    require(_amount>0,"already liquidated");

    if(msg.sender == seller){
        if (invoices[_invoiceId].isFinanceRequested == true &&
invoices[_invoiceId].isPaymentDoneByBuyer == false){
            // reverseSEc case
            msg.sender.transfer(_amount*invoiceDiscountRate/100);
            allowances[payee][msg.sender] = allowances[payee][msg.sender].sub(_amount);
        }
        else{
            //normal business case
            msg.sender.transfer(_amount);
            allowances[payee][msg.sender] = allowances[payee][msg.sender].sub(_amount);
        }
    }
}

```

```

    invoices[_invoiceId].isPaymenttakenBySupplier= true;
    }
    else if(invoices[_invoiceId].isPaymentDoneByBuyer == true) {
    // for banks
    msg.sender.transfer(_amount);
    allowances[payee][msg.sender] = allowances[payee][msg.sender].sub(_amount);
    }
    return true;
}
//-----
-----//

function getpofId(uint _orderId) private view returns(uint pofId){
    for (uint i = 0; i<= totalpof; i++){
        if( poFinances[i].orderId == _orderId){
            return i ;
        }
    }
}

function applyPoFinance(uint _orderId, uint _bankId) public returns(bool success){
    bool _isFinanceRequested = isPOFinanceRequested[_orderId];
    uint _sellerCompanyId =
products[purchaseOrderRequests[_orderId].productId].companyId;
    address seller = companies[_sellerCompanyId].owner;
    uint _buyerCompanyId = purchaseOrderRequests[_orderId].buyerCompanyId;
    require(_isFinanceRequested == false && seller == msg.sender &&
purchaseOrderRequests[_orderId].isAccepted == false
    && invoices[getInvoiceId(_orderId)].invoiceId == 0,"No allowed to apply this POF --
check orderId or bankId");
    uint _qty = purchaseOrderRequests[_orderId].qty;
    uint _price = products[purchaseOrderRequests[_orderId].productId].sellingPrice;
    uint _value = approveReceivables(_qty,_price);
    isPOFinanceRequested[_orderId]= true;
    bool _isApproved = false;
    // poFinanceId = poFinanceId.add(1);
    totalpof = totalpof.add(1);
    poFinances[totalpof] =
PoFinance(totalpof,_bankId,_orderId,_buyerCompanyId,_value,_isApproved,0);
    return true;
}

function getPoFinanceStruct(uint _poFinanceId) private view returns(PoFinance memory
poFinance){
    address poBankAddress = banks[poFinances[_poFinanceId].bankId].bankAddress;

```

```

        require(msg.sender ==
companies[products[purchaseOrderRequests[poFinances[_poFinanceId].orderId].productId].com
panyId].owner
        || msg.sender == poBankAddress,"Access denied to get this POF");
        PoFinance memory poFinance = poFinances[_poFinanceId];
        return(poFinance);
    }

function getPoFinance(uint _poFinanceId) public view returns(
    uint poFinanceId,
    uint bankId,
    uint orderId,
    uint buyerCompanyId,
    uint value,
    bool isApproved,
    uint timeOfApproval
){
    PoFinance memory pofinance = getPoFinanceStruct(_poFinanceId);
    return(
        pofinance.poFinanceId,
        pofinance.bankId,
        pofinance.orderId,
        pofinance.buyerCompanyId,
        pofinance.value,
        pofinance.isApproved,
        pofinance.timeOfApproval
    );
}

function approvePOFinance(uint _poFinanceId) public payable returns(bool success){
    address payable _sellerAddress =
companies[products[purchaseOrderRequests[poFinances[_poFinanceId].orderId].productId].com
panyId].owner;
    address payable _buyerAddress =
companies[purchaseOrderRequests[poFinances[_poFinanceId].orderId].buyerCompanyId].owne
r;
    uint _recievablesValue = poFinances[_poFinanceId].value;
    uint _transferAmount = poDiscountRate*_recievablesValue/100 ;
    address payable _bankAddress = banks[poFinances[_poFinanceId].bankId].bankAddress;
    require ( _bankAddress ==msg.sender && poFinances[_poFinanceId].isApproved == false
&& (msg.value >= _transferAmount),"Approval denied");
    if(msg.value > _transferAmount ) {
        address(uint160(msg.sender)).transfer(msg.value.sub(_transferAmount));
    }
    address(uint160(_sellerAddress)).transfer(_transferAmount);
}

```

```

    poFinances[_poFinanceId].isApproved = true;
    poFinances[_poFinanceId].timeOfApproval = now;

    invoices[getInvoiceId(poFinances[_poFinanceId].orderId)].isPaymenttakenBySupplier =
true;
    emit PoFinanceApproved(_poFinanceId);
    return true;
}

//-----
-----//

function applyApprovedPayableInvoiceFinance(uint _invoiceId, uint _bankId) public returns
(bool success){
    require(isPOFinanceRequested[invoices[_invoiceId].orderId] == false &&
invoices[_invoiceId].isFinanceRequested == false &&
invoices[_invoiceId].isPaymentDoneByBuyer == false &&
    companies[purchaseOrderRequests[invoices[_invoiceId].orderId].buyerCompanyId].owner
== msg.sender && invoices[_invoiceId].isInvoiceFinanceAccepted == true &&
invoices[_invoiceId].isInvoiceApproved==true,"check invoice Id or bank ID or check if the
shipment is confirmed.");
    invoices[_invoiceId].isFinanceRequested = true;
    totalInvoiceFinances = totalInvoiceFinances.add(1);
    uint _buyerCompanyId =
purchaseOrderRequests[invoices[_invoiceId].orderId].buyerCompanyId;
    uint _qty = purchaseOrderRequests[invoices[_invoiceId].orderId].qty;
    uint _price =
products[purchaseOrderRequests[invoices[_invoiceId].orderId].productId].sellingPrice;
    uint _value = approveReceivables(_qty,_price);
    invoiceFinances[totalInvoiceFinances] = InvoiceFinance(totalInvoiceFinances,
_bankId,_invoiceId, _buyerCompanyId, _value, false);
    return true;
}

function approveInvoiceFinance( uint _invoiceFinanceId) public payable returns (bool
success){

    address payable _sellerAddress =
companies[products[purchaseOrderRequests[invoices[invoiceFinances[_invoiceFinanceId].invoi
ceId].orderId].productId].companyId].owner;
    address payable _buyerAddress =
companies[purchaseOrderRequests[invoices[invoiceFinances[_invoiceFinanceId].invoiceId].ord
erId].buyerCompanyId].owner;
    uint _recievablesValue = invoiceFinances[_invoiceFinanceId].value;
    uint _transferAmount = invoiceDiscountRate*_recievablesValue/100 ;

```

```

        require(msg.sender == banks[invoiceFinances[_invoiceFinanceId].bankId].bankAddress
        && invoiceFinances[_invoiceFinanceId].isApproved == false && msg.value >=
        _transferAmount,"approval denied. Check invoice Finance");
        if(msg.value > _transferAmount ) {
            address(uint160(msg.sender)).transfer(msg.value.sub(_transferAmount));
        }
        address(uint160(_sellerAddress)).transfer(_transferAmount);
        invoiceFinances[_invoiceFinanceId].isApproved = true;
        invoices[invoiceFinances[_invoiceFinanceId].invoiceId].isPaymenttakenBySupplier= true;
        allowances[_buyerAddress][_sellerAddress] =
allowances[_buyerAddress][_sellerAddress].sub(_recievablesValue);
        approveRecievables(_buyerAddress,msg.sender,_recievablesValue);
        emit InvoiceFinanceApproved(_invoiceFinanceId);
        return true;
    }

```

```

function getinvoiceFinanceId(uint _invoiceId) private view returns(uint) {
    for (uint i = 0; i<= totalInvoiceFinances; i++){
        if( invoiceFinances[i].invoiceId == _invoiceId){
            return i ;
        }
    }
}

```

```

function getInvoiceFinanceStruct(uint _invoiceFinanceId) private view returns(
InvoiceFinance memory invoiceFinance){
    // uint _invoiceFinanceId = getinvoiceFinanceId(_invoiceId);
    address invoiceFinanceBankAddress =
banks[invoiceFinances[_invoiceFinanceId].bankId].bankAddress;
    require(msg.sender ==
companies[products[purchaseOrderRequests[invoices[invoiceFinances[_invoiceFinanceId].invoi
ceId].orderId].productId].companyId].owner || msg.sender ==
companies[invoiceFinances[_invoiceFinanceId].buyerCompanyId].owner
|| msg.sender == invoiceFinanceBankAddress,"Access denied as Invoice Finance");

    invoiceFinance = invoiceFinances[_invoiceFinanceId];

    return(invoiceFinance );
}

```

```

function getInvoiceFinance(uint _invoiceFinanceId) public view returns(
    uint invoiceFinanceId,
    uint bankId,
    uint invoiceId,

```



```

    uint buyerCompanyId,
    uint value,
    bool isApproved
){
    InvoiceFinance memory invoiceFinance = getInvoiceFinanceStruct(_invoiceFinanceId);
    return(
        invoiceFinance.invoiceFinanceId,
        invoiceFinance.bankId,
        invoiceFinance.invoiceId,
        invoiceFinance.buyerCompanyId,
        invoiceFinance.value,
        invoiceFinance.isApproved
    );
}
//-----
-----//
function registerAsInsuranceCompany( string memory _companyName, string memory
_location, string memory _businessNumber)public {
    uint _companyId = createCompany(_companyName,_location,_businessNumber);
    companies[_companyId].isInsuranceCompany = true;
}

function createSpv(uint _issuerId, uint _insuranceCompanyId) private returns( uint){
    spvId = spvId.add(1);
    spvs[spvId] = SPV(spvId,_issuerId,_insuranceCompanyId,0);
    return spvId;
}

function rateSpv( uint _spvId, uint8 _rating) public {
    require(companies[spvs[_spvId].insuranceCompanyId].owner == msg.sender && _rating
    >=0 && _rating<=5,"Access not given or rating beyond the range.");
    spvs[_spvId].rating = _rating;
}

function getSpv( uint _spvId) public view returns(
    uint spvId,
    uint issuerId,
    uint insuranceCompanyId,
    uint8 rating){
    SPV memory spv = spvs[_spvId];
    return(
        spv.spvId,
        spv.issuerId,
        spv.insuranceCompanyId,
        spv.rating
    )
}

```

```

        );
    }

    function getAllBuyerInvoiceFinancePayable(uint _buyerCompanyId) private returns(uint
_totalfinanciablePayable){
        uint i;
        uint _ownedFinancedAcceptedInvoices =
companyOwnedFinanceAcceptedInvoices[_buyerCompanyId];
        if(_ownedFinancedAcceptedInvoices == 0){
            return 0;
        }
        else{
            for(i=1; i<= totalInvoices; i++){
                if (purchaseOrderRequests[invoices[i].orderId].buyerCompanyId == _buyerCompanyId
&& invoices[i].isInvoiceApproved && invoices[i].isPaymenttakenBySupplier == false &&
invoices[i].isInvoiceFinanceAccepted == true){
                    invoices[i].isFinanceRequested = true;
                    uint _qty= purchaseOrderRequests[invoices[i].orderId].qty;
                    uint _price =
products[purchaseOrderRequests[invoices[i].orderId].productId].sellingPrice;

                    uint _payable = approveReceivables(_qty,_price);
                    _totalfinanciablePayable = _totalfinanciablePayable.add(_payable);
                }
            }
            return _totalfinanciablePayable;
        }
    }

    function applyReverseSecuritization(uint _buyerCompanyId, uint _insuranceCompanyId)
public {
        address _buyerAddress = companies[_buyerCompanyId].owner;
        uint _totalfinanciablePayable = getAllBuyerInvoiceFinancePayable(_buyerCompanyId);
        require(companies[_insuranceCompanyId].isInsuranceCompany ==true &&
_totalfinanciablePayable>0 && _buyerAddress== msg.sender,"the total payable is zero or
insurance company ID is incorrect");
        spvId = createSpv(_buyerCompanyId,_insuranceCompanyId);
        sId = sId.add(1);
        securities[sId] = Security(sId,_totalfinanciablePayable,_buyerCompanyId,spvId);
    }

    function getSecurity( uint _securityId) public view returns (
        uint sId,
        uint amount,
        uint buyerCompanyId,

```

```

    uint spvId
){
    Security memory security = securities[_securityId];
    return(
        security.sId,
        security.amount,
        security.buyerCompanyId,
        security.spvId
    );
}

```

```

function getNoteInfo(uint _noteId) public view returns(
    uint noteId,
    uint sId,
    uint amount,
    address payable owner,
    bool isRedeemed
){
    require(notes[_noteId].owner == msg.sender,"you are not the owner");
    Note memory note = notes[_noteId];

    return(
        note.noteId,
        note.sId,
        note.amount,
        note.owner,
        note.isRedeemed
    );
}

```

```

function buySecurities( uint _sId) public payable{
    // replace _totalSecurityAmount by amount>0 in require
    require(msg.value > 0,"check the value");
    uint _totalSecurityAmount = securities[_sId].amount;
    // how many wei's of securities to buy
    address _buyerAddress = companies[securities[_sId].buyerCompanyId].owner;
    uint _securitiesBought = msg.value*100/invoiceDiscountRate;
    noteId = noteId.add(1);
    if(_securitiesBought > _totalSecurityAmount ) {
        address(uint160(msg.sender)).transfer((msg.value.sub(
        _totalSecurityAmount*invoiceDiscountRate/100)));
        securities[_sId].amount = 0;
        secAllowances[_buyerAddress][msg.sender] =
        secAllowances[_buyerAddress][msg.sender].add(_totalSecurityAmount);
    }
}

```

```

        notes[noteId] = Note(noteId,_sId, _totalSecurityAmount,msg.sender,false);
    }

    else{
        uint _securitiesLeft = _totalSecurityAmount.sub(_securitiesBought);
        securities[_sId].amount = _securitiesLeft;
        secAllowances[_buyerAddress][msg.sender] =
secAllowances[_buyerAddress][msg.sender].add( _securitiesBought);
        notes[noteId] = Note(noteId,_sId, _securitiesBought,msg.sender,false);
    }
}

function liquidateNote ( uint _noteId) public {
    require(notes[_noteId].isRedeemed == false && notes[_noteId].owner ==
msg.sender,"check the note Id or already redeemed");
    address _buyerAddress =
companies[securities[notes[_noteId].sId].buyerCompanyId].owner;
    uint _amount = secAllowances[_buyerAddress][msg.sender];
    require(address(this).balance>= _amount);
    msg.sender.transfer(_amount);
    secAllowances[_buyerAddress][msg.sender] =
secAllowances[_buyerAddress][msg.sender].sub(_amount);
    notes[_noteId].isRedeemed = true;
}

function getRecBalance(address _from) public view returns(uint recbalance){
    return allowances[_from][msg.sender];
}

function getSecBalance(address _from) public view returns(uint){
    return secAllowances[_from][msg.sender];
}
// function() external payable {
//     // a fallback function
// }
}

```

Appendix C RegisterCompanyInterface.sol

```
pragma Solidity >=0.5.0 <0.6.9;

interface RegisterCompanyInterface{

    function createCompany( string calldata _companyName, string calldata _location, string
calldata _businessNumber) external returns(uint);

    function getCompany (uint _companyId) external view returns (
        uint companyId,
        address payable owner,
        string memory companyName,
        string memory location,
        string memory businessNumber,
        bool isInsuranceCompany
    );

    function registerProduct( uint _companyId, string calldata _prodName,uint _sellingPrice )
external returns(bool);

    function getProduct (uint _productId ) external view returns (
        uint productId,
        string memory prodName,
        uint companyId,
        uint sellingPrice
    );
}
```

Appendix D CompleteFinancingContractInterface

```
pragma Solidity >=0.5.0 <0.6.0;
```

```
interface CompleteFinancingContractInterface{
```

```
    function manufactureProd(uint _productId, uint _amount) external returns(bool);
```

```
    function getInventory(uint _productId) external view returns(uint);
```

```
    function removeProduct(uint _productId) external returns (bool success);
```

```
    function registerBank(string calldata _bankName, string calldata _bankCode )external returns  
(bool success);
```

```
    function getBank(uint _bankId) external view returns (  
        uint bankId,  
        string memory bankName,  
        string memory bankCode,//confirm with prof  
        address payable bankAddress);
```

```
    function makePurchaseOrderRequest(uint _productId, uint _qty, uint _buyerCompanyId )  
external returns (bool success);
```

```
    function getPurchaseOrderRequest(uint _orderId) external view returns(  
        uint purchaseOrderId,  
        uint productId,  
        uint buyerCompanyId,  
        uint qty,  
        bool isAccepted  
    );
```

```
    function acceptOrder( uint _orderId, uint _carrierId, uint _paymentDueDays, bool  
_invoiceFinanceAccepted) external returns (bool success);
```

```
    function getInvoice(uint _orderId) external view returns(  
        uint invoiceId,  
        uint orderId,  
        uint paymentdeadline,  
        bool isInvoiceApproved,  
        bool isPaymentDoneByBuyer,  
        bool isPaymenttakenBySupplier,  
        bool isFinanceRequested,
```

```

    bool isInvoiceFinanceAccepted,
    uint timeOfCreation
);

function getBillofLading( uint _orderId) external view returns(
    uint bolId,
    uint orderId,
    uint buyerCompanyId,
    uint carrierId,
    string memory carrierName
);

function getShipment( uint _orderId) external view returns(
    uint trackingId,
    uint carrierId,
    string memory deliveryAddress,
    uint qty,
    bool isShipmentRecieved
);

function confirmShipment(uint _invoiceId) external returns (bool success);

function cancelOrder(uint _orderId) external returns(bool success);

function makePayment( uint _invoiceId) external payable returns (bool success);

function LiquidateReceivables( uint _invoiceId) external returns (bool success);

////////////////////////////////////

function applyPoFinance(uint _orderId, uint _bankId) external returns(bool success);

function getPoFinance(uint _poFinanceId) external view returns(
    uint poFinanceId,
    uint bankId,
    uint orderId,
    uint buyerCompanyId,
    uint value,
    bool isApproved,
    uint timeOfApproval
);

function approvePOFinance(uint _poFinanceId) external payable returns(bool success);

```

```
////////////////////////////////////
```

```
function applyApprovedPayableInvoiceFinance(uint _invoiceId, uint _bankId) external returns  
(bool success);
```

```
function approveInvoiceFinance( uint _invoiceFinanceId) external payable returns (bool  
success);
```

```
function getInvoiceFinance(uint _invoiceFinanceId) external view returns(  
    uint invoiceFinanceId,  
    uint bankId,  
    uint invoiceId,  
    uint buyerCompanyId,  
    uint value,  
    bool isApproved  
);
```

```
////////////////////////////////////
```

```
function registerAsInsuranceCompany( string calldata _companyName, string calldata  
_location, string calldata _businessNumber) external ;
```

```
function rateSpv( uint _spvId, uint8 _rating) external;
```

```
function getSpv (uint _spvId) external view returns (  
    uint spvId,  
    uint issuerId,  
    uint insuranceCompanyId,  
    uint8 rating);
```

```
function applyReverseSecuritization(uint _buyerCompanyId, uint _insuranceCompanyId)  
external;
```

```
function getSecurity( uint _securityId) external view returns (  
    uint sId,  
    uint amount,  
    uint buyerCompanyId,  
    uint spvId  
);
```

```
function getNoteInfo(uint _noteId) external view returns(  
    uint noteId,  
    uint sId,  
    uint amount,  
    address payable owner,
```



```
        bool isRedeemed
    );

    function buySecurities( uint _sId) external payable;

    function liquidateNote ( uint _noteId) external;

    function getRecBalance(address _from) external view returns(uint recbalance);

    function getSecBalance(address _from) external view returns(uint secBalance);

}
```

Appendix E SafeMath.sol

```
pragma Solidity ^0.5.0;

library SafeMath {
    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
        require(c >= a, "SafeMath: addition overflow");

        return c;
    }

    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        require(b <= a, "SafeMath: subtraction overflow");
        uint256 c = a - b;

        return c;
    }

    function mul(uint256 a, uint256 b) internal pure returns (uint256) {
        if (a == 0) {
            return 0;
        }

        uint256 c = a * b;
        require(c / a == b, "SafeMath: multiplication overflow");

        return c;
    }

    function div(uint256 a, uint256 b) internal pure returns (uint256) {
        require(b > 0, "SafeMath: division by zero");
        uint256 c = a / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold

        return c;
    }

    function mod(uint256 a, uint256 b) internal pure returns (uint256) {
        require(b != 0, "SafeMath: modulo by zero");
        return a % b;
    }
}
```

Appendix F Contract Structure

<p><u>Business</u></p> <p><u>Structs</u> PurchaseOrderRequest Invoice BillOfLading Shipment</p> <p><u>Mappings</u> purchaseOrderRequests invoices billOfLadings shipments productToInventory</p> <p><u>Variables</u> orderID totalInvoices totalBoID trackingID</p> <p><u>Functions</u> manufactureProd getInventory removeProd calculateRecievables approveRecievables makePurchaseOrderRequest getPurchaseORderRequests acceptOrder makeInvoice makeBillOfLading confirmShipment getInvoice getBillOfLading getShipment cancelOrder makePayment LiquidateRecievables</p>	<p><u>Purchase Order Finance</u></p> <p><u>Structs</u> Bank PoFinance</p> <p><u>Mappings</u> POFinances isPoFinanceApplied Banks ownedBanks</p> <p><u>Variables</u> poFinanceID poDiscountRate</p> <p><u>Functions</u> registerBank getBank getpoID applyPoFinance getPoFinance approvePOFinance makePayment liquidateRecievables getRecBalance</p>
<p><u>Reverse Factoring</u></p> <p><u>Structs</u> Bank InvoiceFinance</p> <p><u>Mappings</u> invoiceFinances banks companyOwnedFinanceAcceptedInvoices</p> <p><u>Variables</u> invoiceFinanceID invoiceDiscountRate</p> <p><u>Functions</u> applyApprovedPayableInvoiceFinance getInvoiceFinanceID getInvoiceFinanceStruct getInvoiceFinance approveInvoiceFinance makePayment liquidateRecievables getRecBalance</p>	<p><u>Reverse Securitization</u></p> <p><u>Structs</u> Spv Security Note</p> <p><u>Mappings</u> spvs securiteis notes</p> <p><u>Variables</u> slID spvID noteID invoiceDiscountRate</p> <p><u>Functions</u> makePayment liquidateRecievables registerInsuranceCompany createSpv rateSpv getSpv getAllBuyerInvoiceFinancePayable applyReverseSecuritization getSecurity getNoteInfo buySecurities liquidateNote getSecBalance getRecBalance</p>

Appendix G Use Cases

G.1 Use case 1

COMPILER ⓘ

0.5.17+commit.d19bba13 ⌵

☐ Include nightly builds

LANGUAGE

Solidity ⌵

EVM VERSION

compiler default ⌵

COMPILER CONFIGURATION

☐ Auto compile

☒ Enable optimization

☐ Hide warnings

Img G.1 Compilation Details Remix

ADDRESS	BALANCE	TX COUNT	INDEX	
0x9EeB5FC5758EA1EDE3d2Bf005cb956EaD8b975b4	100.00 ETH	0	0	🔑
ADDRESS	BALANCE	TX COUNT	INDEX	
0x77BBC44322a8de787e20f4F0F3f3548e757e3a81	100.00 ETH	0	1	🔑
ADDRESS	BALANCE	TX COUNT	INDEX	
0xA2EC0E179982214b60C4960b57Cc521D7dbE47b9	100.00 ETH	0	2	🔑
ADDRESS	BALANCE	TX COUNT	INDEX	
0x740ee7cAfeF1C1026d8383AeD714977601A99f1D	100.00 ETH	0	3	🔑
ADDRESS	BALANCE	TX COUNT	INDEX	
0xbD37467F27D03fA96F5A77F1574aDc4dE12469CA	100.00 ETH	0	4	🔑
ADDRESS	BALANCE	TX COUNT	INDEX	
0x73CbE18D6E8Cb058C53CB49d8a5f3014b7fCc9C5	100.00 ETH	0	5	🔑
ADDRESS	BALANCE	TX COUNT	INDEX	
0xf6eff7279E7a744de2B2c4d6821fffF602E16d4E2	100.00 ETH	0	6	🔑

Img G.2 Test Accounts in Ganache

TX HASH	CONTRACT CREATION		
0x7f7e45f04eb4a728b2538f0dadcb5326f1a2a2f163bfaf44c0cd8486ddf359ba			
FROM ADDRESS	CREATED CONTRACT ADDRESS	GAS USED	VALUE
0x9EeB5FC5758EA1EDE3d2Bf005cb956EaD8b975b4	0x8aCDD0dB2A780A2bc69707FdBE633F88A4CA60285	5864970	0

Img G.3 Contract Creation

TX HASH 0x518695b8e349a80943d9cbe03ef9e2766dfc1180f500d4a4fdab22e306016ac3				CONTRACT CALL
FROM ADDRESS 0x73CbE18D6E8Cb058C53CB49d8a5f3014b7fCc9C5	TO CONTRACT ADDRESS 0x8aCDDd82A780A2bc69707FdB6E633F88A4CA60285	GAS USED 166616	VALUE 0	
TX HASH 0xac4d1852d7c89e54cb8ce4ac45e887b43a157be69c9300919adb4f7e027d1ea1				CONTRACT CALL
FROM ADDRESS 0xbD37467F27D03FA96F5A77F1574aDc4dE12469CA	TO CONTRACT ADDRESS 0x8aCDDd82A780A2bc69707FdB6E633F88A4CA60285	GAS USED 186906	VALUE 0	
TX HASH 0x6d473b65f51e1059f2bec9b8e292fa47397919ed9365679bb73d00a84efaa6dc				CONTRACT CALL
FROM ADDRESS 0x740ee7cAfeF1C1026d8383AeD714977601A99F1D	TO CONTRACT ADDRESS 0x8aCDDd82A780A2bc69707FdB6E633F88A4CA60285	GAS USED 170220	VALUE 0	
TX HASH 0x538f281246ec2440aefcbe4483d1fc2c427b3850d08297afa49701a9e2e810f4				CONTRACT CALL
FROM ADDRESS 0xA2EC0E179902214b60C4960b57Cc521D7dbE47b9	TO CONTRACT ADDRESS 0x8aCDDd82A780A2bc69707FdB6E633F88A4CA60285	GAS USED 166360	VALUE 0	
TX HASH 0x1d57cd4c28aa375cfe139be55193dd76e5d96f9056cf72739ddfc5e05793585b				CONTRACT CALL
FROM ADDRESS 0x77BBc44322a8de787e20f4f0F3f3548e757e3a81	TO CONTRACT ADDRESS 0x8aCDDd82A780A2bc69707FdB6E633F88A4CA60285	GAS USED 166104	VALUE 0	
TX HASH 0x5573508babf246d69188c6bc80a62cdd84644505404bdcc146c79a10d3871a9b				CONTRACT CALL
FROM ADDRESS 0x9EeB5FC5758EA1EDE3d28f05cb956Ea08b975b4	TO CONTRACT ADDRESS 0x8aCDDd82A780A2bc69707FdB6E633F88A4CA60285	GAS USED 196488	VALUE 0	

Img G.4 Onboarding Transactions

GETCOMPANY	5	
0:	uint256: companyId	5
1:	address: owner	0x73CbE18D6E8Cb058C53CB49d8a5f3014b7fCc9C5
2:	string: companyName	supplier2
3:	string: location	supplier2
4:	string: businessNumber	5
5:	bool: isInsuranceCompany	false

Img G.5 Company Details Queried from Blockchain

GETPRODUCT	1	
0:	uint256: productId	1
1:	string: prodName	product1
2:	uint256: companyId	1
3:	uint256: sellingPrice	1000000000000000000

Img G.6 Product Details Queried from Blockchain

GETINVENTORY	1	
0:	uint256: 2	

Img G.7 Supplier Successfully Queried Inventory

GETPURCHASEORDE... 1	0: uint256: purchaseOrderId 1 1: uint256: productId 1 2: uint256: buyerCompanyId 2 3: uint256: qty 2 4: bool: isAccepted true 5: bool: isFinanceRequested false
GETBILLOFLADING 1	0: uint256: bolId 1 1: uint256: orderId 1 2: uint256: buyerCompanyId 2 3: uint256: carrierId 3 4: string: carrierName carrier
GETSHIPMENT 1	0: uint256: trackingId 1 1: uint256: carrierId 3 2: string: deliverAddress buyer 3: uint256: qty 2 4: string: receiverAddress buyer 5: bool: isShipmentReceived false
GETINVOICE 1	0: uint256: invoiceId 1 1: uint256: orderId 1 2: uint256: paymentDeadline 30 3: bool: isInvoiceApproved false 4: bool: isPaymentDoneByBuyer false 5: bool: isPaymentTakenBySupplier false 6: bool: isFinanceRequested false 7: bool: isInvoiceFinanceAccepted true 8: uint256: timeOfCreation 1595297199

Img G.8 Purchase Order, Invoice, Bill of Lading and Shipment Successfully Queried

GETRECBALANCE	0x77BBC44322a8de787e20f4F0F3f3548e757e3a81
0:	uint256: recbalance 200000000000000000

Img G.9 Supplier's REC Balance After Buyer Confirms Shipment

TX HASH		CONTRACT CALL	
0x9b55397ecf6aca3e8a3b11f28cb33e479b861e380ea58f005eef92441056f277			
FROM ADDRESS	TO CONTRACT ADDRESS	GAS USED	VALUE
0x77BBC44322a8de787e20f4F0f3f3548e757e3a81	0x8aCDDd82A780A2bc69707FdBE633F88A4CA60285	31300	200000000000000000
ADDRESS		BALANCE	TX COUNT
0x77BBC44322a8de787e20f4F0F3f3548e757e3a81		97.99 ETH	4
		INDEX	
		1	

Img G.10 Successful Payment by the Buyer

TX HASH		CONTRACT CALL	
0xf89e867e412ba4ded21521ba11587e111f697fbd3c5b00bb4acb5ebcf2bc2f33			
FROM ADDRESS	TO CONTRACT ADDRESS	GAS USED	VALUE
0x9EeB5FC5758EA1EDE3d2Bf005cb956EaD8b975b4	0x8aCDDd82A780A2bc69707F0BE633F88A4CA60285	27863	0
ADDRESS	BALANCE	TX COUNT	INDEX
0x9EeB5FC5758EA1EDE3d2Bf005cb956EaD8b975b4	101.87 ETH	6	0


Img G.11 Successful Ether Withdrawal by the Seller

G.2 Use Case 2


GETPURCHASE...	2
0:	uint256: purchaseOrderId 2
1:	uint256: productId 1
2:	uint256: buyerCompanyId 2
3:	uint256: qty 1
4:	bool: isAccepted false
5:	bool: isFinanceRequested true

GETINVOICE	2
0:	uint256: invoiceId 0
1:	uint256: orderId 0
2:	uint256: paymentDeadline 0
3:	bool: isInvoiceApproved false
4:	bool: isPaymentDoneByBuyer false
5:	bool: isPaymentTakenBySupplier false
6:	bool: isFinanceRequested false
7:	bool: isInvoiceFinanceAccepted false
8:	uint256: timeOfCreation 0

Img G.12 Successful Querying of Purchase Order and Invoice

ADDRESS	BALANCE	TX COUNT	INDEX	
0x9EeB5FC5758EA1EDE3d2Bf005cb956EaD8b975b4	102.67 ETH	46	0	

Img G.13 Supplier Recieves Ether after POF approval

ADDRESS	BALANCE	TX COUNT	INDEX	
0x740ee7cAfeF1C1026d8383AeD714977601A99f1D	100.14 ETH	20	3	

Img G.14 Bank Successfully Withdraws Ether after Buyer Pays the Invoice

G.3 Use Case 3

GETINVOICE	3
0:	uint256: invoiceId 3
1:	uint256: orderId 3
2:	uint256: paymentDeadline 30
3:	bool: isInvoiceApproved true
4:	bool: isPaymentDoneByBuyer false
5:	bool: isPaymentTakenBySupplier false
6:	bool: isFinanceRequested true
7:	bool: isInvoiceFinanceAccepted true
8:	uint256: timeOfCreation 1595439363

Img G.15 Invoice generated Accepting Invoice Finance

ADDRESS	BALANCE
0x9EeB5FC5758EA1EDE3d2Bf005cb956EaD8b975b4	103.55 ETH
ADDRESS	BALANCE
0x77BBC44322a8de787e20f4F0F3f3548e757e3a81	96.94 ETH
ADDRESS	BALANCE
0xA2EC0E179982214b60C4960b57Cc521D7dbE47b9	99.98 ETH
ADDRESS	BALANCE
0x740ee7cAfeF1C1026d8383AeD714977601A99f1D	99.24 ETH

Img G.16 Account Balances and REC Balance after Bank approves Invoice Finance

SECURITIES	SPVS
0: uint256: sld 1	0: uint256: spvld 1
1: uint256: amount 4000000000000000000	1: uint256: issuerId 2
2: uint256: buyerCompanyId 2	2: uint256: insuranceCompanyId 4
3: uint256: spvld 1	3: uint8: rating 5

GETNOTEINFO	1	GETNOTEINFO	2
0:	uint256: noteld 1	0:	uint256: noteld 2
1:	uint256: sld 1	1:	uint256: sld 1
2:	uint256: amount 222222222222222222	2:	uint256: amount 1777777777777777778
3:	address: owner 0xf6eff7279E7a744de2B2c4d6821ff602E16d4E2	3:	address: owner 0x33d3E6929F93E1366f674eDAc07224A32a9C2174
4:	bool: isRedeemed false	4:	bool: isRedeemed false

ADDRESS	BALANCE
0x9EeB5FC5758EA1EDE3d2Bf005cb956EaD8b975b4	105.33 ETH

ADDRESS	BALANCE
0x73CbE18D6E8Cb058C53CB49d8a5f3014b7fCc9C5	101.77 ETH

Img G.19 Both the Seller Withdraw Ethers from the Contract

ADDRESS	BALANCE
0xf6eff7279E7a744de2B2c4d6821ffF602E16d4E2	100.22 ETH

ADDRESS	BALANCE
0x33d3E6929F93E1366f674eDAc07224A32a9C2174	100.18 ETH

Img G.20 Both the Investors Withdraw Ethers from the Contract

Appendix H CompleteFinancingTest.js

```
const CompleteFinancingContract = artifacts.require("CompleteFinancingContract");
require('chai')
.use(require('chai-as-promised'))
.should();

contract("CompleteFinancingContract", (accounts) => {
  // defining the accounts of the GanacheCLI
  let[supplier1,supplier2,buyer1,
buyer2,bank1,bank2,insurance1,insurance2,investor1,investor2] = accounts;
  let contractInstance;
  beforeEach(async () => {
    contractInstance = await CompleteFinancingContract.deployed();
  })

  describe("CORRECT DEPLOYMENT.", async () =>{
    it("should return Contract Address", async() =>{
      const contractAddress = await contractInstance.contractAddress();
      assert.notEqual(contractAddress, "", "address is not empty");
      assert.notEqual(contractAddress,null,"address is not null");
      assert.notEqual(contractAddress,undefined,"address is defined");
    })

    it("should return correct token names", async() => {
      const REC = await contractInstance.REC();
      const SEC = await contractInstance.SEC();
      assert.equal(REC,"REC for Recievables");
      assert.equal(SEC,"SEC for Securities");
    })
  })

  describe("BASIC BUSINEE OPERATIONS MUST BE FLUID ", async() => {
    describe("MUST ONBOARD ACTORS AND PRODUCTS CORRECTLY.", async() =>{
      it("should onboard suppliers correctly ---> createCompany and GetCompany must work",
      async()=> {
        result = await contractInstance.createCompany("supplier","supplier", "1",{from:
supplier1 });
        result2 = await contractInstance.getCompany("1", {from:supplier1 });
        const event = result.logs[0].args;
        assert.equal(event.companyId.toNumber(),1,"companyId is right and event can be
read");
        assert.equal(event.name,"supplier","Name is right and event can be read" );
        assert.equal(result2.owner,supplier1,"owner is right and info can be fetched!");
      })
    })
  })
})
```

```

    assert.equal(result2.location, "supplier", "location is right! getCompany is working!!");
  })

  it(" should allow suppliers to register product and update inventory", async() => {
    result = await contractInstance.registerProduct("1","product1","10000000000000000000",{from:supplier1});
    result2 = await contractInstance.getProduct("1",{from:buyer1});
    result3 = await contractInstance.manufactureProd("1","2",{from:supplier1});
    await contractInstance.manufactureProd("1","2",{from:supplier2}).should.be.rejected;
    result4 = await contractInstance.getInventory("1",{from:supplier1});
    await contractInstance.getInventory("1",{from:supplier2}).should.be.rejected;
    await contractInstance.removeProduct("1",{from:supplier2}).should.be.rejected;
    result5 = await contractInstance.removeProduct("1",{from:supplier1});
    result6 = await contractInstance.getProduct("1",{from:supplier2});
    const event = result.logs[0].args;
    assert.equal(event.productId.toNumber(),1,"productId is right and event can be read");
    assert.equal(event.prodName,"product1","product name is right.");
    assert.equal(result2.sellingPrice, "10000000000000000000","get Product is working");
    assert.equal(result4,"2","manufactureProd is working.");
    assert.equal(result6.productId,0,"removeProduct working.")
  })

  it("should onboard Bank and Insurance Company",async() => {
    result = await contractInstance.registerBank("bank","bank",{from:bank1})
    result2 = await contractInstance.getBank("1",{from:buyer1});
    assert.equal(result2.bankName,"bank","correct bank Name");
    assert.equal(result2.bankAddress,bank1,"correct Address");
    result3 = await contractInstance.registerAsInsuranceCompany("Insurance","Insurance","4",{from:insurance1})
    result4 = await contractInstance.getCompany("2", {from: supplier2});
    assert.equal(result4.companyName, "Insurance", "correct name");
    assert.equal(result4.isInsuranceCompany,true,"registered as an insurance company")
  })
})

describe ( "MUST ENABLE ORDERING, ACCEPTING AND PAYMENT",async() => {

  it("should be able to make order request and accept it", async() => {
    await contractInstance.registerProduct("1","product1","10000000000000000000",{from:supplier1});
    await contractInstance.manufactureProd("2","2",{from: supplier1});
    await contractInstance.createCompany("buyer","buyer", "3",{from: buyer1});
    await contractInstance.createCompany("carrier","carrier", "4",{from: buyer2});
    await contractInstance.makePurchaseOrderRequest("2","2","3",{from:buyer1});
    await contractInstance.getPurchaseOrderRequest("1",{from:supplier2}).should.be.rejected;
  })
})

```

```

        result = await contractInstance.getPurchaseOrderRequest("1",{from: supplier1 })
        assert.equal(result.productId,"2","correct product ID requested for purchase");
        assert.equal(result.isAccepted,false,"PO not accepted yet");
        await
contractInstance.acceptOrder("1","4","30",true,{from:supplier2}).should.be.rejected;
        await contractInstance.acceptOrder("1","4","30",false,{from:supplier1 });
        await contractInstance.getInvoice("1",{from:bank1 }).should.be.rejected;
        result2 = await contractInstance.getInvoice("1",{from:buyer1 });
        result3 = await contractInstance.getBillofLading("1",{from:buyer1 });
        assert.equal(result3.carrierId,"4","correct carrier is chosen");
        assert.equal(result3.buyerCompanyId,"3","correct delivery address")
        assert.equal(result.isAccepted,false,"PO not accepted yet");
        assert.equal(result2.orderId.toNumber(),1,"correct order number");
        assert.equal(result2.isPaymentDoneByBuyer, false, "Invoice is not yet accepted");
    })

    it("buyer should be able to make payment and seller should be able to accept the payment
", async() => {
        await
contractInstance.makePayment("1",{from:supplier2,value:2000000000000000000}).should.be.re
jected;
        await
contractInstance.makePayment("1",{from:buyer1,value:2000000000000000000}).should.be.reje
cted;
        await contractInstance.confirmShipment("1",{from:buyer1 });
        result = await contractInstance.getShipment("1",{from:supplier1 });
        assert.equal(result.isShipmentRecieved,true,"shipment correctly delivered")
        assert.equal(result.deliveryAddress,"buyer","delivered at the correct address")
        await
contractInstance.makePayment("1",{from:buyer1,value:2000000000000000000});
        const contractAddress = await contractInstance.contractAddress();
        await contractInstance.getInvoice("1",{from:buyer1 });
        result1 = await contractInstance.getInvoice("1",{from:buyer1 });
        assert.equal(result1.isPaymentDoneByBuyer,true, "invoice payment successfully
done");
        let sellerOldBalance;
        sellerOldBalance = await web3.eth.getBalance(supplier1);
        sellerOldBalance = parseInt(sellerOldBalance);
        let totalRecievables;
        totalRecievables = await contractInstance.getRecBalance(buyer1, {from:supplier1 });
        totalRecievables = parseInt(totalRecievables);
        await contractInstance.LiquidateReceivables("1",{from: supplier2}).should.be.rejected;
        liquidateRec      =      await      contractInstance.LiquidateReceivables("1",{from:
supplier1 }).should.not.be.rejected;
    })

```

```

    })
  })
  describe("FINANCING SHOULD WORK", async()=>{
    it("seller should be able to apply and bank should be able to accept POF", async()=>{
      await contractInstance.registerBank("bank2","bank2",{from:bank2});
      result = await contractInstance.getCompany("4",{from:supplier1});
      await contractInstance.makePurchaseOrderRequest("2","2","3",{from:buyer1});
      await contractInstance.applyPoFinance("2","2",{from:supplier2}).should.be.rejected;
      await contractInstance.applyPoFinance("2","2",{from:supplier1});
      await contractInstance.getPoFinance("1",{from:bank1}).should.be.rejected;
      result = await contractInstance.getPoFinance("1",{from:bank2});
      assert.equal(result.orderId, "2","orderID is right");
      assert.equal(result.buyerCompanyId,"3","buyer company Id is right")
      assert.equal(result.isApproved, false, "PO is correctly not approved")
      assert.equal(parseInt(result.value,10),2000000000000000000);
      let sellerOldBalance;
      sellerOldBalance = await web3.eth.getBalance(supplier1);
      sellerOldBalance = parseInt(sellerOldBalance);
      await
bank1,value:2000000000000000000}).should.be.rejected;
      await
bank2,value:2000000000000000000});
      let sellerNewBalance = await web3.eth.getBalance(supplier1);
      sellerNewBalance = parseInt(sellerNewBalance);
      let expectedBalance = sellerOldBalance + 1600000000000000000;
      result1 = await contractInstance.getPoFinance("1",{from:bank2});
      assert.equal(result1.isApproved, true, "PO finance is correctly approved.");
      assert.equal(sellerNewBalance,expectedBalance);
      await contractInstance.manufactureProd("2","2",{from: supplier1});
      await contractInstance.acceptOrder("2","4","30",true,{from:supplier1});
      await contractInstance.confirmShipment("2",{from:buyer1});
      await contractInstance.makePayment("2",{from:buyer1,value:2000000000000000000});
      liquidateRec      =      await      contractInstance.LiquidateReceivables("2",{from:
bank2}).should.not.be.rejected;
    })

    it("should allow buyer to apply invoiceFinance and bank to accept it",async() => {
      // will reject Invoice Financing if seller specifies false
      await contractInstance.makePurchaseOrderRequest("2","2","3",{from:buyer1});
      await contractInstance.manufactureProd("2","2",{from: supplier1});
      await contractInstance.acceptOrder("3","4","30",false,{from:supplier1});
      await contractInstance.confirmShipment("3",{from:buyer1});
      await
contractInstance.applyApprovedPayableInvoiceFinance("3","1",{from:buyer2}).should.be.reject
ed;

```

```

        await
contractInstance.applyApprovedPayableInvoiceFinance("3","1",{from:buyer1}).should.be.rejected;

        //buyer has to do normal payment
        await contractInstance.makePayment("3",{from:buyer1,value:2000000000000000000});
        await
supplier1}).should.not.be.rejected;
        contractInstance.LiquidateReceivables("3",{from:
        await contractInstance.makePurchaseOrderRequest("2","2","3",{from:buyer1});
        await contractInstance.manufactureProd("2","2",{from: supplier1});
        await contractInstance.acceptOrder("4","4","30",true,{from:supplier1});
        await contractInstance.confirmShipment("4",{from:buyer1});
        await contractInstance.applyApprovedPayableInvoiceFinance("4","1",{from:buyer1});
        result = await contractInstance.getInvoiceFinance("1",{from:bank1})
        assert.equal(result.invoiceFinanceId,1, "InvoiceFinance ID is correct.");
        assert.equal(result.isApproved,false, "Correctly not yet approved.");
        let sellerOldBalance;
        sellerOldBalance = await web3.eth.getBalance(supplier1);
        sellerOldBalance = parseInt(sellerOldBalance);
        await
contractInstance.approveInvoiceFinance("1",{from:bank2,value:2000000000000000000}).should
d.be.rejected;
        await
contractInstance.approveInvoiceFinance("1",{from:bank1,value:2000000000000000000});
        result1 = await contractInstance.getInvoiceFinance("1",{from:bank1})
        assert.equal(result1.isApproved,true, "correctly approved.");
        let value = parseInt(result.value);
        let expectedBalance = sellerOldBalance + value*90/100;
        let sellerNewBalance = await web3.eth.getBalance(supplier1);
        sellerNewBalance = parseInt(sellerNewBalance);
        assert.equal(sellerNewBalance,expectedBalance, "seller sent money");
        await contractInstance.makePayment("4",{from:buyer1,value:2000000000000000000});
        liquidateRec      =      await      contractInstance.LiquidateReceivables("4",{from:
bank1}).should.not.be.rejected;
    })

    it("should allow buyer to securitize the invoices and investors to buy the securities issued",
    async() => {
        await contractInstance.makePurchaseOrderRequest("2","2","3",{from:buyer1});
        await contractInstance.createCompany("supplier2","supplier2", "5",{from: supplier2});
        await
contractInstance.registerProduct("5","product1","2000000000000000000",{from:supplier2});
        await contractInstance.makePurchaseOrderRequest("3","2","3",{from:buyer1});
        await contractInstance.manufactureProd("2","2",{from: supplier1});
        await contractInstance.manufactureProd("3","2",{from: supplier2});
        await contractInstance.acceptOrder("5","4","30",true,{from:supplier1});

```

```

        await contractInstance.acceptOrder("6","4","30",true,{from:supplier2});
        await contractInstance.confirmShipment("5",{from:buyer1});
        await contractInstance.confirmShipment("6",{from:buyer1});
        await
contractInstance.applyReverseSecuritization("3","2",{from:buyer2}).should.be.rejected;
        await contractInstance.applyReverseSecuritization("3","2",{from:buyer1});
        result = await contractInstance.getSecurity("1",{from:investor1});
        assert.equal(result.sId, "1","correct security ID");
        assert.equal(result.amount,"6000000000000000000","securities correctly issued");
        assert.equal(result.spvId,"1","correct SPV")
        await contractInstance.rateSpv("1","4",{from: insurance2}).should.be.rejected;
        await contractInstance.rateSpv("1","4",{from: insurance1});
        result1 = await contractInstance.getSpv("1",{from:investor1});
        assert.equal(result1.rating,"4","correctly rated by insurance company.")
        await
contractInstance.buySecurities("1",{from:investor1,value:4000000000000000000});
        await
contractInstance.buySecurities("1",{from:investor2,value:4000000000000000000});
        await
contractInstance.LiquidateReceivables("5",{from:supplier1}).should.not.be.rejected;
        await
contractInstance.LiquidateReceivables("6",{from:supplier2}).should.not.be.rejected;
        await contractInstance.getNoteInfo("1",{from:investor2}).should.be.rejected;
        result3 = await contractInstance.getNoteInfo("1",{from:investor1});
        result4 = await contractInstance.getNoteInfo("2",{from:investor2});
        assert.equal(result3.owner,investor1,"owner is correct");
        assert.equal(result4.noteId,"2","note ID is correct");
        assert.equal(result3.isRedeemed,false,"correctly not redeemed");
        assert.equal(result4.isRedeemed,false,"correctly not redeemed");
        await contractInstance.makePayment("5",{from:buyer1,value:2000000000000000000});
        await contractInstance.makePayment("6",{from:buyer1,value:4000000000000000000});
        await contractInstance.liquidateNote("1",{from:investor2}).should.be.rejected;
        await contractInstance.liquidateNote("1",{from:investor1}).should.not.be.rejected;
        await contractInstance.liquidateNote("2",{from:investor2}).should.not.be.rejected;
    })
})
})

```