Generating Trusted Coordination of Collaborative

Software Development Using Blockchain

by

Jinal S. Patel

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved July 2020 by the
Graduate Supervisory Committee:

Stephen S. Yau, Chair
Ajay Bansal
Jia Zou

ARIZONA STATE UNIVERSITY

August 2020

ABSTRACT

The coordination of developing various complex and large-scale projects using computers has been well established and is the so-called computer-supported cooperative work (CSCW). Collaborative software development consists of a group of teams working together to achieve a common goal for developing a high-quality, complex, and large-scale software system efficiently, and it requires common processes and communication channels among these teams. The common processes for coordination among software development teams can be handled by similar principles in CSCW. The development of complex and large-scale software becomes complicated due to the involvement of many software development teams. The development of such a software system can be largely improved by effective collaboration among the participating software development teams at both software components and system levels. The efficiency of developing software components depends on trusted coordination among the participating teams for sharing, processing, and managing information on various participating teams, which are often operating in a distributed environment. Participating teams may belong to the same organization or different organizations. Existing approaches to coordination in collaborative software development are based on using a centralized repository to store, process, and retrieve information on participating software development teams during the development. These approaches use a centralized authority, have a single point of failure, and restricted rights to own data and software. In this thesis, the generation of trusted coordination in collaborative software development using blockchain is studied, and an approach to achieving trusted cooperation for collaborative software development using

blockchain is presented. The smart contracts are created in the blockchain to encode software specifications and acceptance criteria for the software results generated by participating teams. The blockchain used in the approach is a private blockchain because a private blockchain has the characteristics of providing non-repudiation, privacy, and integrity, which are required in trusted coordination of collaborative software development. This approach is implemented using Hyperledger, an open-source private blockchain. An example to illustrate the approach is also given.

DEDICATION

This thesis is dedicated to my father, Sunil Patel, my sisters Pragati Patel and Princy Patel, and Viraj Talaty wholeheartedly, to offer me constant unconditional love and support. Thank you for encouraging me during my graduate studies with all help and inspiration to succeed.

ACKNOWLEDGMENTS

I would like to express my earnest appreciation to my advisor Dr. Stephen S. Yau, for his encouragement and guidance throughout my thesis. He was always patient with me to give useful insights and feedback on my research. I would like to thank Dr. Yau for giving me the opportunity to do research under his guidance. I would also like to express gratitude to my graduate supervisory committee members, Dr. Ajay Bansal and Dr. Jia Zou, to agree to serve on the committee.

I sincerely thank my colleagues, Chandralekha Yenugunti, Suli Adeniye, and Brandon Anderson, in Dr. Yau's research laboratory for their friendship and numerous valuable discussions of the research.

TABLE OF CONTENTS

Page

LIST OF TABLES vi

LIST OF FIGURES vii

CHAPTER

1 INTRODUCTION 1

2 BACKGROUND 4

2.1 Collaborative Software Development 4

2.2 Blockchain 7

3 PROBLEM STATEMENT 21

4 THE OVERALL APPROACH TO GENERATING TRUSTED COORDINATION OF COLLABORATIVE SOFTWARE DEVELOPMENT . 23

5 GENERATION OF SMART CONTRACTS 27

5.1 Software Design Smart Contract 32

5.2 Software Testing Smart Contract 42

5.3 Software Requirement Smart Contract 48

6 ILLUSTRATIVE EXAMPLE OF THE APPROACH 56

7 EVALUATION 63

8 CONCLUSION AND FUTURE WORK 66

REFERENCES 67

BIOGRAPHICAL SKETCH 69

LIST OF TABLES

Table Page

1. The Function Level Privileges for Smart Contract Methods 31

2. The Software Component Specifications of the Example Project 60

3. A Summary of the Important Properties of Our Approach, Centralized, and Decentralized Approaches for Generating Trusted Coordination of Collaborative Software Development...64

LIST OF FIGURES

Figure Page

1. A Centralized Tool or Repository to Handle Coordination in Collaborative Software Development 5

2. A Decentralized Tool for Collaborative Software Development....7

3. A Blockchain Infrastructure 9

4. The Structure of the Block in Blockchain 10

5. Blockchain Initialization for N Participating Software Development Teams....24

6. The Software Activities Performed by Each Participating Team in the Approach 25

7. The Smart Contract for Example 1 33

8. The Smart Contract for Example 2 43

9. The Smart Contract for Example 3 50

10. The Software Development Activities Required for Each Software Component....56

11. The Functional Dependency Graph of the Major Software Components in the Example Project....58

CHAPTER 1

INTRODUCTION

Multiple teams participate in software development activities to achieve high quality and quick software delivery for a complex and large-scale software system. The coordination of developing complex and large-scale projects using computers has been well established and is the so-called computer-supported cooperative work (CSCW) [1]. Coordination becomes necessary in multiple ways in collaborative software development, such as in software specifications sharing, software data sharing, to understand software components with past data, and to verify proposed techniques for acceptance. The collaborative software development requires similar principles like CSCW to handle the coordination among participating software development teams. In the absense of a coordination environment, each participating team will likely to manage its software development activities on its own. Each team can have different data and software sharing policies, different tools to maintain information related to software, different development environments to carry out coordination. In such an environment, complex and large-scale software development becomes chaos due to heterogeneous tools, development environments, and delay in coordination among teams. These factors cause trust issues among participating teams in collaborative software development. In the development of complex and large-scale software, each software development phase needs input from the previous software development phase and passes essential data to the next software development phase. This information builds a history of the software component, which is a crucial piece of information to audit in the collaborative software development environment. Trusted coordination is necessary for all kinds of software development

paradigms, such as waterfall, spiral, or agile. Each team is coordinating with other teams directly if they need to collect information on any part of software development activities. Easy availability of complete and reliable auditable history of software components is difficult to obtain in centralized approaches [2-7]. Since the sharing information of software components among development teams is quite common in a complex and large-scale software development project, a participating team cannot always trust the information on shared components provided by the other participating teams, especially when some participants may belong to competitors or possibly malicious groups involved in the collaboration [8][9]. The involvement of third-party vendors can create more severe trust issues because of different data sharing policies on software development and quality of software development. Hence the trust among the participating teams will be reduced due to the involvement of many participating teams.

The concerns on the trust issues can be significantly reduced by adequately utilizing blockchain [10] technology, which has properties of transparency, reliability, and auditability in data and software specification sharing. In this thesis, the generation of trusted coordination in collaborative software development using blockchain in presented. The smart contracts are created in the blockchain to encode software specifications and acceptance criteria for the software results generated by participating teams. The blockchain used in the thesis is a private blockchain because a private blockchain has the characteristics of providing non-repudiation, privacy, and integrity, which are required in the trusted coordination of collaborative software development. An approach to generating trusted coordination of collaborative software development is presented and implemented

using Hyperledger, an open-source private blockchain. An example to illustrate the approach is also presented in this thesis.

The organization of the thesis is presented in seven chapters. The first chapter provides an overview of the research on collaborative software development and the organization of the thesis document. The second chapter provides the necessary background to understand the research problem in collaborative software development and blockchain. The third chapter describes the problem statement and limitations of a few approaches in the current state of the art. The fourth chapter explains the blockchain-based overall approach in detail. Chapter five presents an approach to generate a smart contract and few sample smart contracts for different phases of the software development lifecycle. In the sixth chapter, an illustrative example is shown with a sample complex and large-scale software development project. Chapter seven presents the discussion and conclusion of the approach. Chapter eight gives a brief overview of the future work on the blockchain-based approach.

CHAPTER 2

BACKGROUND

2.1 Collaborative Software Development

Software development becomes a complex process due to the increasing demand for software in all possible domains. Software development comprises multiple software activities such as software requirement engineering, design, implementation, testing, and maintenance, and these activities may be performed by various teams based on the complexity and necessity of the software development project. Multiple teams need to come together to develop a complex and large-scale software system to produce a high-quality software system. Collaboration helps software development teams to produce software quickly. Software development teams are adopting multiple paradigms to carry out software development activities, such as the waterfall model, spiral model, or agile methodologies. All these paradigms require multiple teams to coordinate to achieve their final goal. Participating teams in the collaborative software development project may belong to the same organization or different organization. A team collaborates with other teams to share software specifications, software results, and communication for other software development information. Various collaborative software development tools are developed to facilitate coordination among participating teams [2-7]. The collaborative software development environment has many advantages, such as teams can use past software information to audit or verify software components. The participating teams in the collaboration may distribute effort and share results for common components to lower cost and improve software quality. Teams can save efforts of software development significantly by sharing software component specification and its history, including

different phases of the software development lifecycle. The success of the complex and large-scale software development project relies on how well different participating teams are working collaboratively. All participating teams need to come to a consensus on useful tools, technologies, methodologies for an efficient and reliable software development environment. Various tools are available opensource, and on commercial platforms to manage coordination among software development teams. These tools use a centralized repository to share, process, and manage information regarding participating teams, as shown in Figure 1.

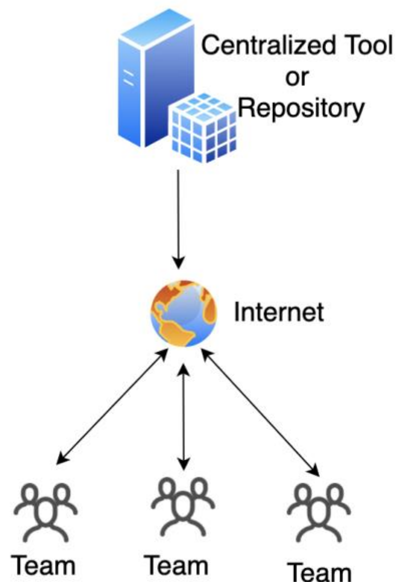


Figure 1: A Centralized Tool or Repository to Handle Coordination in Collaborative Software Development

Centralized tools are useful to store information on different phases of software development activities and create a history of the software component. Software developers may use the history of the component to verify the reliability, efficiency, and usefulness of the software component. Furthermore, the software development team can significantly reduce the cost and build time for the new software component development

by utilizing results and history available for the past software component. Collaborative software development has shown promising results in the current demand for reliable and fast software development. There are few advantages of using centralized tools for collaborative software development, such as lower hardware resources, single-point authority, easy to change, easy to scale.

Centralized tools for the coordination in collaborative software development have the following limitations, use of a centralized authority, have a single point of failure, and restricted rights to own data and software. Due to current threats to software systems, every day, we see a new cyber-attack launched on such systems. As per recent study regarding cyber-attacks on software systems, 8,854 breaches are recorded between January 1, 2005, and April 18, 2018 [11]. Many commercial tools became a victim of such data breaches. Additionally, centralized tools do not provide a feature of data restrictions on the sharing of critical software component specifications and results. Due to these weaknesses, trust among participating teams may be reduced.

To overcome limitations on centralized tools, various decentralized and distributed tools, as shown in Figure 2, are developed to manage coordination in collaborative software development. The most popular open-source distributed version control tool for source code tracking is GitHub [12], and GitHub provides a platform for software developers to develop software collaboratively and in distribution fashion. However, when one takes an in-depth look into GitHub, it is a repository to store content with version control capability. GitHub provides few necessary features like immutable commit history, parallel workflow for software development projects, and pluggable merge strategies. Distributed tools like GitHub has the following limitation in collaborative software development, lack of access

management capability, mutable history, no consensus criteria for adding data into the tool. GitHub was attacked several times for arbitrary code executions and distributed denial of service (DDoS) in the past.

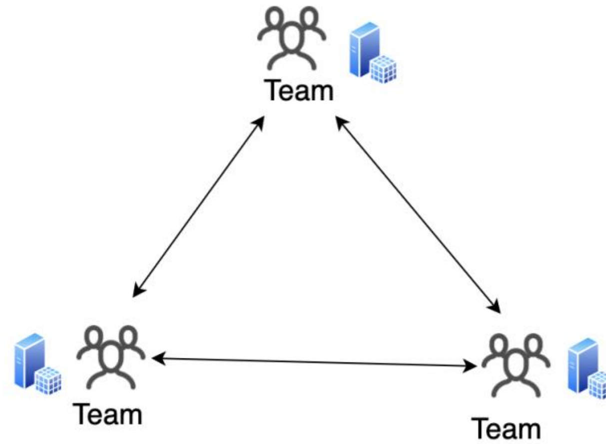


Figure 2: A Decentralized Tool for Collaborative Software Development

2.2 Blockchain

Blockchain is one of the expanding emerging technologies, which uses cryptographic techniques to build and maintain ever-growing immutable records in a distributed fashion. The Blockchain was first introduced in 2009 through Bitcoin's initiation [13]. Blockchain eliminates the need for centralized authority due to its distributed nature. As blockchain's name implies, it is a chain of blocks containing transactions. The transaction can be a record of tangible assets or intangible assets. Properties, cars, hardware can be considered as Tangible assets (having a physical form). Patents, intellectual properties, and copyrights can be considered as Intangible assets (having no physical form). Both kinds of assets can be recorded in the blockchain. Blockchain is a kind of advanced data structure, where blocks are connected with

cryptographic links, and it is an encrypted ledger of transactions. Blockchain is tamper-evident and tamper-resistant [14], which means it provides an immutable distributed ledger with an auditability feature. Blockchain was widely connected with a cryptocurrency like bitcoin [13] due to its origin, but the underlying technology is being used in many other applications.

Blockchain is a complex technology to understand and implement for a specific application. The following terms should be understood in detail to understand blockchain technology for applicability of any approach:

- Blockchain network
- Block structure
- Cryptographic functions
- Consensus Protocols

Blockchain network

Blockchain network is the infrastructure of multiple nodes that provides distributed immutable ledger. A participant electronic device is called a node in the Blockchain. These nodes can store a complete copy of the blockchain ledger or partial copy based on the participation of the node in Blockchain [14]. If a node stores the complete copy, then it is called a Major node, and it is responsible for becoming online all the time to get the latest copy of the Blockchain. If a node stores a partial copy of the ledger, then it is called a minor node. The more major node joins the blockchain network, the better the tamper-resistant it becomes. When the blockchain data spread across many devices, it will be challenging for a hacker to update past data stored in the Blockchain. The sample infrastructure for the blockchain infrastructure is shown in Figure 3.

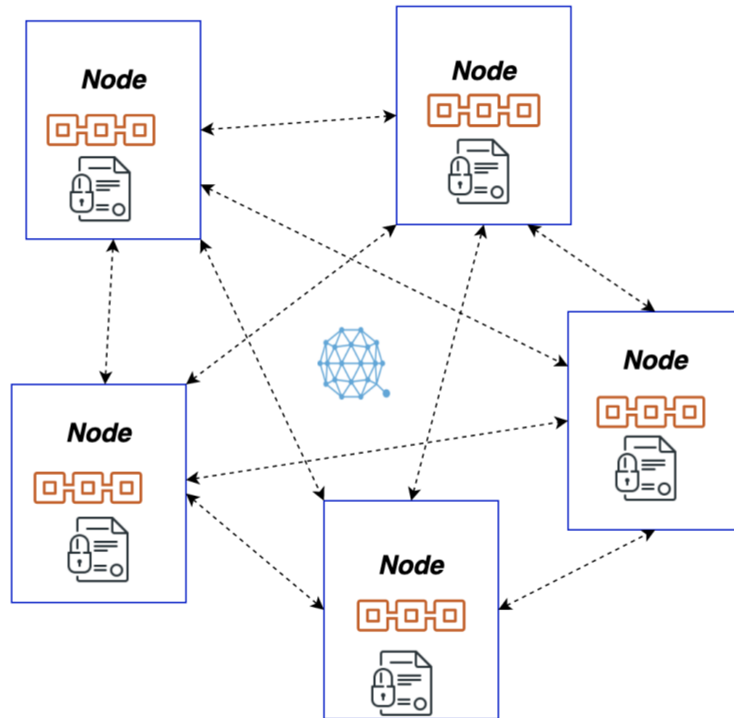


Figure 3: A Blockchain Infrastructure

Block Structure

Each block in the blockchain has two components: Block header and Block body. Block header stores various information on the block, such as block number, block version, Merkle tree root, previous block's hash, timestamp, size of the block, and nonce [14]. The Block body consists of blockchain transactions to be added to the blockchain. These transactions are not necessarily in sequence; it can be picked at any order. The Block structure is shown in Figure 4 with a block header and block body. Different parts of the block header are as follows:

Block Number: Block number is the current height of the blockchain, how many blocks are already added in the blockchain plus one.

Block Version: Version of the blockchain.

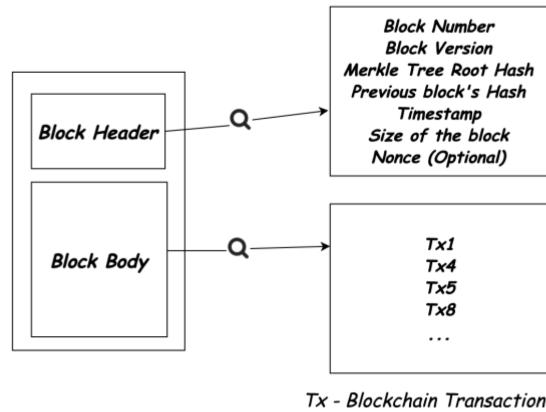


Figure 4: The Structure of the Block in Blockchain

Merkle Tree Root Hash: In the blockchain, block data is hashed in the form of a Merkle tree root hash. Merkle tree is a data structure to store data in a way that a single root is found by hashing and combining data stored in leaf nodes [15]. In the blockchain, Merkle tree root hash is computed by hashing and combining transactions in the block body. Merkle tree root hash is then stored in the header of the block and used for providing features of tamper-resistance of the blockchain. If anyone changes data in the block, then Merkle tree root hash will be different from one stored in the block header. This data structure is useful to verify the consistency and integrity of the data stored in the blockchain.

Previous Block's Hash: The hash of the complete header of the previous block is stored in the latest block. Cryptographic chain the blockchain is created by storing previous block's hash in the current block. This property creates a linked list like structure with linked blocks. The first block in the blockchain has no previous block, so it stores value 0 as the previous block's hash. Previous block's hash is also used for providing features of tamper-resistance of the blockchain. If anyone changes data in the previous block, then computed

hash for the block will be different from one stored in the next block's header. This data is useful to verify the consistency and integrity of the data stored in the blockchain.

Timestamp: The current time in seconds in the format of the universal time since January 1, 1970, when the current block is created.

Size of the Block*:* stores size of the block, including block header and block body.

Nonce: Nonce is the abbreviation for the Number Used Only Once. A nonce is used in the blockchain, which uses a consensus model "proof of work." Only changing the nonce value provides a mechanism in blockchain mining for obtaining different digest values while keeping the same block data [14]. This digest value needs to be less than the target hash value.

Target Hash: Target hash is a difficulty level from 0 to 2^{256} number, which is initialized in the genesis block. When a new block is added to a blockchain, the hash number H_n of the new block is computed as follows:

if $H_n <$ Target Hash:

add a new block in the blockchain

else:

reject new block

Block Body: List of transactions stored in the current block. These transactions are picked randomly, and it is not mandatory to have sequential transactions.

Genesis Block: The beginning block in the blockchain is called Genesis Block. It is a special block in the blockchain, which stores necessary configurations for the blockchain. Genesis block records the initial state of the blockchain [14] with configurations such as consensus protocol, target hash (difficulty level), block reward policy (optional), and rate of blockchain creation.

Cryptographic Functions

Hash functions: Hash function is a one-way and irreversible function that takes any size of the input and outputs it to fixed-sized message digest [14]. Example hash functions are the MD family (MD4, MD5), SHA family (SHA-0, SHA-1, SHA-256, SHA-512), SCrypt, BCrypt. One of the cryptographic hash function is SHA-256, which takes any arbitrary size of the input and map that to 256-bit size output. The output of the hash functions is generally rendered in hexadecimal form. Cryptographic hash functions are used in data integrity, to protect sensitive data, in the verification of digital signatures, and in the application of TLS, SSL, IPsec. The hash functions frequently used in blockchain platforms are SHA-256 and SCrypt.

Asymmetric Key Pairs: Blockchain uses asymmetric key pairs, which is also known as public-key cryptography. Asymmetric key pairs are one of the vital components in blockchain technology. Public key cryptography utilizes a pair of keys (Public Key, Private

Key) for each participating user. The public key from the pair is broadcasted to all other members of the blockchain. The private key is a secret key for the node, and only known to the owner of the key. Public key cryptography is used for a digital signature on blockchain transactions and private communication channel setup. Due to the usage of two keys in asymmetric cryptography, data is encrypted using one key and decrypted with others. It is a safer way to make sure that only allowable user receives key for communication. In the blockchain, each blockchain transaction is signed with the private key of the owner before adding that into the blockchain. Once it is added into the blockchain, anyone with the knowledge of the public key of the owner can verify the owner's signature.

Digital Signatures: In the real world, handwritten signatures are used to bind signatory to the handwritten or typed message. Similarly, digital signatures are used to bind signatory to digital documents or messages. Digital signatures use public-key cryptography in the blockchain. Digital signatures are used for data integrity, transaction authentication, and non-repudiation. For example, one node signed a blockchain transaction with its private key and added it to the blockchain. Any node in the blockchain can verify this blockchain transaction using the node's public key. Few well-known signing algorithms are RSA-based signature schemes, Digital Signature Algorithm (DSA), and Elliptic Curve Digital Signature Algorithm (ECDSA) [14].

Consensus Protocols/Models

Blockchain uses different consensus protocols to verify the addition of new blocks in the blockchain. The consensus is agreement criteria or conflict resolution technique defined in the genesis block. Following is a detailed explanation of the consensus protocols used in the blockchain, and a brief overview of the advantages and disadvantages of each consensus protocol.

Proof of Work (PoW): In proof of work model, a user can add a new block in the blockchain by solving a cryptographic puzzle. The cryptographic solving puzzle involves a resource-intensive process of finding a Nonce value, which is less than the target hash defined in the genesis block. The target value may be adjusted (up or down) to regulate how blocks are being published frequently [14]. The proof of work model is ideal for blockchains with nodes with less trust among them. This puzzle is designed in a way that solving puzzles is resource-intensive and challenging, but verifying the solution of the puzzle is very easy and intuitive. Available computing resources and time increases over time to enhance the difficulty of the puzzle. Finding nonce is a brute force method, and miner needs to try different nonce value to match the target hash value. Whichever node solves the puzzle first, that node broadcast nonce value with block data to all other nodes. Other nodes in the blockchain verify the solution and accept the result. If the nonce is accepted by other nodes, then the new block is appended to the blockchain. This entire process is known as blockchain mining. The user receives a reward for solving the cryptographic puzzle in the form of cryptocurrency. Bitcoin and Ethereum use proof of

work consensus model. Proof of work model is vulnerable to a classic 51% attack on the blockchain, where the majority of nodes collaborate to solve cryptographic puzzles [14].

Proof of Stake (PoS): Proof of stake model uses the stake of the user in the blockchain mining process. Proof of stake was created as a replacement to Proof of work model in public blockchains. The user gets the reward in proportion to the stake he/she holds in the blockchain. Instead of utilizing intensive resources and computing power to solve PoW, a PoS miner is entitled to mine a percentage of transactions based on his or her ownership stake. If a user has a 5% stake in blockchain mining, he will get 5% of the reward when a new block gets created. Proof of stake generates a situation where 51% attack does not give many advantages to the attacker because if someone already has a 51% share, they are not going to attack the network with this share. The value of an attacker's cryptocurrency will be impacted if they perform such attacks. Ethereum is planning to switch to blockchain uses PoS consensus.

Round Robin: Permissioned blockchain networks mostly use the round-robin consensus model. Just like the round-robin algorithm, nodes in the blockchain network take a turn to create the next block in the blockchain [14]. A time limit may be imposed to handle situations where publishing node is not available to publish the next block in its turn. Round robin model does not require intensive computing power or mining process like proof of work and proof of stake models. Each node takes a turn to create blocks; hence no node can create a majority of the nodes in this approach. This model requires nodes to trust each other to create nodes in turns, so permissionless or public blockchain networks cannot use this model due to trust issues among participants.

Proof of Authority/Identity: Proof of Authority consensus model is also known as Proof of Identity consensus model. This model needs publishing nodes to attach their real identity with it, so that node can be identified quickly and trace back to the human user. This model needs some trust in the blockchain system, where publishing nodes verifies and prove their identity before joining the blockchain network [14]. This model is widely used along with the reputation system. The reputation of the publishing node analyzed from its constant behavior in the blockchain. If any node behaves suspiciously, it is implausible that block from that node gets published. The proof of identity model is more suitable for permissioned blockchain.

Practical Byzantine Fault Tolerance (pBFT): PBFT consensus model is based on the voting system among participating blockchain nodes. Publishing node needs to get two-third of the votes to publish the next blockchain node. pBFT provides promising results even if one-third of the nodes are malicious in the blockchain. This model is more suitable for permissioned blockchain, where the identity of each node is known and verified before joining the network.

Blockchain Classifications

Blockchain networks are categorized into two types based on the identification model of the participating nodes. These two types are public/permissionless blockchain and private/permissioned blockchain.

Public Blockchain: If anyone with the internet can participate in the blockchain network, then it is called public blockchain or permissionless blockchain. Anyone in the blockchain can add and verify a new block in the blockchain. Node does not require any prior permission to join public blockchain; it is more like working on an open-source project [14]. Establishing trust among blockchain nodes is difficult in public blockchain because any node on the internet can join blockchain with enough computing power and resources. Blockchain nodes are incentivized by any form of cryptocurrency in this model. Bitcoin and Ethereum are the popular blockchain platforms to use public blockchain. Most public blockchains are resource and computing-intensive due to the use of cryptocurrency and block mining process.

Private Blockchain: If a node needs permission to participate in the blockchain network, then it is called private blockchain or permissioned blockchain. Private blockchain has a partial trust established among blockchain nodes due to their pre-verified identity; it is more like an enterprise network with known entities [14]. The private blockchain is not resource-intensive like public blockchain. Private blockchains can provide read-write restrictions for participating in the blockchain [14]. Various consensus models are used in private or permissioned blockchain to establish trust in publishing nodes.

Smart Contracts

Nick Szabo presented the notion of a smart contract in 1994 and defined a smart contract as “a transaction treaty that executes the terms of a contract automatically” [17]. Smart contracts are executable code in the blockchain environment and allow the maintenance of

transactions without third parties in the blockchain. The primary goal of the smart contract is to provide contractual conditions, handle its exceptions in the blockchain [17]. A smart contract contains asset data and executable code in the blockchain. The addition of the smart contract and each execution of a smart contract is stored in the blockchain as a transaction. Bitcoin and Ethereum blockchain platforms use smart contract functionality, and a node needs to spend cryptocurrency to execute this smart contract, whereas private blockchain like Hyperledger has the same functionality as chaincode without the need of any cryptocurrency.

Traits of Blockchain

Distributed: Nodes in the blockchain are connected through distributed fashion. Each node saves a complete or partial copy of the blockchain transactions. Data tampering becomes difficult due to the distributed design of the blockchain networks.

Decentralized: Nodes in the blockchain need to come to a consensus to add any new block in the blockchain-based on the consensus model they use; this feature eliminates the need for centralized authority.

Immutability: Blockchain has a hype of being immutable, but actually, blockchain is tamper-evident and tamper-resistant [14] due to cryptographic link between two nodes, and presence of Merkle tree root hash in each node header. Modifying any past transaction is difficult and requires so many cryptographic updates in the blockchain. Additionally, the distributed nature of the blockchain makes it nearly impossible to update previously added data.

Consensus: Consensus models decide who will add a new block in the blockchain. Nodes can participate in the blockchain without necessarily trusting each other due to the power of consensus models and creates a trustless environment.

Auditability: Each node can verify all the transactions in the blockchain. This feature provides a node to validate or audit previous entries and provide transparent history for future use.

Blockchain has the hype of being an emerging technology with many promising characteristics. Though blockchain has many advantages due to its nature and cryptocurrency, one needs to carefully consider the benefits and limitations of using blockchain in its approach. Blockchain has several limitations like storage requirement, resource consumption, scalability, throughput, and latency of the transaction processing times.

There is a debate going on the topic of the use of a shared repository using GitHub vs. blockchain. Blockchain and GitHub are fundamentally different in their implementation. Github uses a tree structure, where hash data is stored in a tree structure with direct access to its predecessor. Blockchain uses the structure of LinkedList with cryptographic links and Merkle tree root hash for block data. Blockchain links all the data stored in blockchain from the very first block, and anyone can verify this data in the blockchain. Blockchain implementation requires verification of each newly added block by participating node based on various consensus models, and GitHub does not require verification of added data by multiple participants. Hence blockchain can be used in a case where multiple parties are collaborating with trust issues. Git provides facility to rewrite past data, delete existing data from GitHub, which is against the immutability feature of

the blockchain. Git also uses the cryptographic hash to store individual commit history, which provides characteristics of tamper-evident and tamper-resistance like blockchain. The significant difference between two technology is working structure; blockchain uses one chain to store it's all data, where git uses different branches to store data, which will cause multiple forks in blockchain implementation. Lastly, blockchain implementation is distributed by nature, where git can be centralized or distributed as per the need of the user.

CHAPTER 3

PROBLEM STATEMENT

In a collaborative software development environment, participating teams working with other teams to achieve larger goals of greater importance and generate better quality software [9]. So far, existing approaches to the coordination in collaborative software development [9] [13] [17-22] are for developing a single complex and large-scale software system. Some existing approaches [19-21] use a shared repository to manage various versions of the software component and the dependency among their software components [19][20] and show how to monitor software quality matrices in collaborative software development [21]. Trust management is a critical issue in the complex and large-scale software development project, considering the current trend of increasing threats to software systems with important applications. With multiple teams and organizations involved in developing a complex and large-scale software project, the shared repository and tool for the collaboration must provide transparency, auditability, and reliability to handle software components without concerns about the trust issues.

Furthermore, shared repository and tool for collaboration should have the capability of sharing data through some private communication channels to protect certain critical information in software development and to provide data ownership to some participating teams as needed. If outsourced software or open-source software is used in the software development project, then it should be easy for all the participating teams to know the history of the outsourced or opensource software, if needed. In case a centralized solution for coordination in collaborative software development is used, concerns such as single point of failure, restricted data ownership, lack of verification for the data to be added,

private sharing of critical data, data tampering, and data auditability, may be raised. These concerns may be serious for collaborative software development.

As mentioned before, some approaches have used blockchain to verify blockchain transactions in collaborative software development, and do not address the coordination in collaborative software development. In this approach, blockchain is used to provide trusted coordination for collaboratively developing a complex and large-scale software system.

CHAPTER 4

THE OVERALL APPROACH TO GENERATING TRUSTED COORDINATION OF COLLABORATIVE SOFTWARE DEVELOPMENT

In this section, the overall approach to generating trusted coordination in collaborative software development is presented. A blockchain-based approach is implemented using a private or permissioned blockchain for developing a complex and large-scale software system in a collaborative environment.

In the approach, a blockchain represents the entire collaborative software development project. Each participating software development team is represented by a distinct node in the blockchain. Only one of the participating teams is designated as the prime contractor team, which is responsible for negotiating and determining all the responsibilities of each participating team in the software development project. Participating teams may belong to different organizations or the same organization as the primary contractor. All software development activities are performed in a distributed way except role assignment and blockchain setup.

The approach is implemented in private blockchain to accommodate enterprise setup for collaborative software development. Each node in the blockchain is directly connected to other nodes, and this structure creates a complete graph of N nodes. Assume N be the number of nodes in the blockchain used in the approach. The initialization of the blockchain is shown in Figure 5. Each node saves an entire copy of the blockchain, which satisfy the consensus criteria of the blockchain. Each node has a designated member of the team as the administrator of the team to provide input and output of the software development activities to/from the team. Practical Byzantine Fault Tolerance (PBFT)

consensus model is used in the approach to collect agreement of participating nodes in a collaborative software development project.

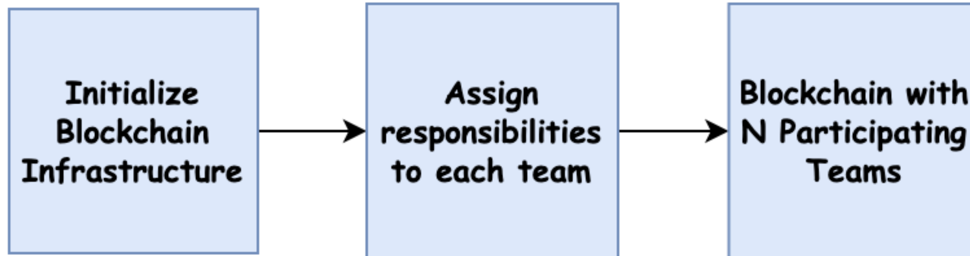


Figure 5: Blockchain Initialization for N Participating Software Development Teams

Following are the notations used in the approach:

- T: A participating team in the collaborative software development project.
- N: The number of T's in the collaborative software development project.
- C: A smart contract generated by a T.
- M: The number of software specifications, each of which will generate a smart contract by a T in the collaborative software development project.
- R: A software result represented as a transaction in the blockchain.
- S: The number of R's in the blockchain.
- G: The group of all the recipient teams for a smart contract.

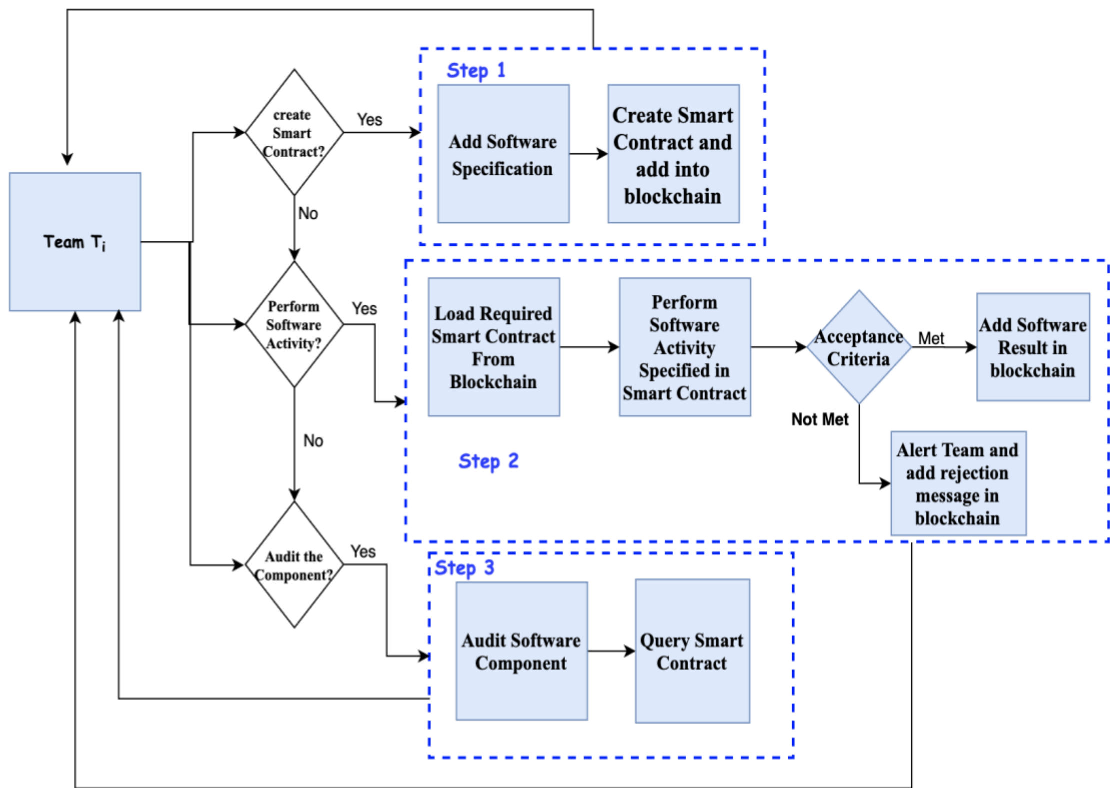


Figure 6: The Software Activities Performed by Each Participating Team in the Approach

The following steps describe the overall approach performed by each participating teams:

Step 1: $\forall T_i \mid i = 1, 2, \ldots N$, generates up to M software specifications, each in the form of a smart contract $C_{i,k}$, from T_i to T_k, $k \neq i$, $T_k \in G_i$, and G_i is the group of all recipient teams of $C_{i,k}$. Each $C_{i,k}$ contains all the required specifications for the software developed by T_k, and satisfying the acceptance criteria of $C_{i,k}$.

Repeat this process for $\forall T_i$, $i = 1, 2, \ldots N$. The generation of the smart contract will be presented in Chapter 5.

Step 2: $\forall T_k \in G_i$, T_k performs the software activity specified in $C_{i,k}$ generated in Step 1.

Step 2.1: If the result R_t ,$1 \leq t \leq S$, generated by Tk meets the acceptance criteria of $C_{i,k}$, R_t is added to the blockchain as a blockchain transaction.

Step 2.2: If R_t generated by T_k does not meet the acceptance criterion of $C_{i,k}$, T_i is alerted by the blockchain that R_t is not successful, and added in the blockchain with the rejection message.

Repeat Step 2, $\forall T_k \in G_i$.

Transaction in Step 2.1 and 2.2 is added in the blockchain if it meets consensus criteria defined for the blockchain.

Step 3: If a T_i, $1 \leq i \leq N$, requires to audit the development history of a software component, T_i verifies all the R_t's $1 \leq t \leq S$ by querying $C_{i,k}$, $k \neq i$, $Tk \in Gi$.

The software activities performed by each participating team in the approach are shown in Figure 6.

CHAPTER 5

GENERATION OF SMART CONTRACTS

Smart contracts are executable code in the blockchain environment and allow the process of transactions without third parties in the blockchain. The primary goal of the smart contract is to provide contractual conditions, handle its exceptions in the blockchain [17]. A smart contract contains asset data and executable code in the blockchain but not include the information on how the transaction is processed or originated. The approach to generating trusted coordination of collaborative software development is implemented on the Hyperledger – open-source private blockchain. Hyperleger [23] supports many programming languages, including GoLang (go), java, NodeJS. The blockchain-based approach uses go for the smart contract implementation because Hyperledger's internal development is done in go, and API for the go is more matured than other supported languages. Also, using go is useful to set up or modify an existing open-source Hyperledger platform for customization.

In this section, a process for generating smart contracts for software development specifications is presented with few examples. Smart contracts are executable code in the blockchain environment and maintain transactions without third parties in the blockchain. In the blockchain-based approach, each software team generates multiple software specifications in the form of smart contracts for each phase of the software development lifecycle. Each smart contract contains the following significant parts for software specification:

a. Allowed Parties: All teams who can participate in this Smart Contract.

b. Contract Terms and Metadata: Required information to perform software activity and acceptance criteria for software specification in {Key, Value} format. Each {Key, Value} pair is being checked against the generated result by other allowed teams.

c. Execution Terms for Result Compliance: team setup execution action for acceptance or rejection of the supplied software result.

The creation of the Smart Contract require to implement the following major steps:

Step1): create Structure (class or struct) with the following information:

- AllowedParties (type list – holds recipient teams)

  Example: [T_1, T_2, T_3] (Assign responsibility of each team in software requisites)

- ContractMetadata (type map – holds key-value pairs)

  Example: Owner: (variable type - String)

  SoftwareModule: (variable type - Number)

  Version: (variable type - String)

  CreationDate: (variable type Date)

  SourceLocation: (variable type String)

- Software Requisites (variable type map: key-value pairs)
  - Key: Software specification conditions
  - Value: Expected value/s
  - Example:

    CyclicDependency: 0 (in class and package diagrams) ⇒Assigned to which participating team

ExternalLibraries: [4-8] (number of external libraries allowed to be used in the code) ⇒ Assigned to which participating team

…

Step 2): Write implementation for the following two default methods to create a smart contract.

- Init – to initialize the smart contract
- Invoke – to call any method from the smart contract after initialization

Following are the pseudocodes for the above default methods in the blockchain-based approach:

```
function Init
    # instantiate Structure created in Step 1)
    struct SoftwareSpecs


function Invoke(String functionName, String attributeName, var attributeValue, String softwareResult)
    if functionName = "query"
        return SoftwareSpecs [attributeName]
    else if functionName = "audit"
        return "all transactions related to this smart contract"
    else if functionName = "update"
        SoftwareSpecs [attributeName] = attributeValue
        # update attribute in the blockchain
    else if functionName = "delete"
```

```
        # delete the smart contract
    else if functionName = “processResult”
        if softwareResult = SoftwareSpecs[attributeName]
            # update blockchain with software result and success message
        else
            alert SoftwareSpecs[‘Owner’] through email
             # update blockchain with an error message
```

Note that if a smart contract is deleted from the blockchain, all the data input, output for the smart contract will remain in the blockchain. In addition, the blockchain-based approach is able to handle the dynamic nature of changing requirements in a smart contract through the “update” and “delete” method. The approach requires each team to create a separate smart contract for each major software versions. Minor versions are handled through updating acceptance criteria in the existing smart contract through the “update” and “delete” method.

The method to generate a smart contract is common for all phases of software development in a collaborative software development project. Each software development phase does not mandate information from preceding or previous software development phases. In a nutshell, a smart contract for collaborative software development contains data (data and metadata of a software component) and actions (methods to perform different activities on software data). Each method in the smart contract is given certain privileges as per the requirement of the software component to prevent the deletion and modification

of smart contract attributes by malicious participants. Table 1 presents function level privileges for each method presented in the smart contract.

Method	Function level access assignment
Init	Owner of the smart contract
Invoke query	Any participant on the blockchain
Invoke audit	Any participant on the blockchain
Invoke update	Owner of the smart contract
Invoke delete	Owner of the smart contract
Invoke processResult	All participants in the smart contract
Invoke alert	All participants in the smart contract including Owner

Table 1: The Function Level Privileges for Smart Contract Methods

Three smart contracts are presented in this section to show how software development teams can generate smart contracts for various phases of the software development lifecycle. The smart contracts cover methods to generate a smart contract for the various phase of the software development lifecycle, as mentioned in Section 5.1. Software design and software testing smart contracts cover the quantifiable assessments of software results, and software requirement covers non-quantifiable assessments of the software result based on requirement traceability matrix. The generation of the smart contract for the requirement phase is complicated than other phases, and require more details, so design phase, and testing phase is presented in Section 5.1 and 5.2, and the generation of smart contract for requirement phase is presented in 5.3. The structure of the presentation does not change actual sequence of software development activities for any software development phases. The method to generate a smart contract for the remaining phases are equivalent to these three examples and presented in Section 5.

5.1 Software Design Smart Contract

A smart contract for the software design phase of a collaborative software development project, which includes the necessary information to perform software design assessment, and acceptance criteria for software design evaluation. Assessment matrices for the software design phase are decided based on the input from the previous phases, the requirement of the software component, and by collaborating software development teams. Software design can be assessed using several matrices, like the dependency of each class in a class diagram, package dependencies, cycle dependency in a class diagram, and abstractness of each class in the example. There are many assessment matrices available for the software design phase, and the selection of the matrix is decided by the participating team in the design phase. The participating teams decide which assessment matrix is suitable for that component. Encoding all possible matrices in a smart contract is beyond the scope of this thesis. Participating team decides consensus criteria from pBFT and round-robin based on the requirement of the software component and relation among participating teams. Following the above process Step 1) in Section 5.1 input (a, b, and c) are generated for the smart contract shown in Figure 7.

According to Step 2) in Section 5 following methods are generated and implemented to carry out software activity mentioned in the smart contract for software design phase:

- Init: Instantiate structure or class with required software specification metadata. Sample struct information is shown below with sample values required in the software design phase:

  Sample code in go programming language is shown as follows for the smart contract:

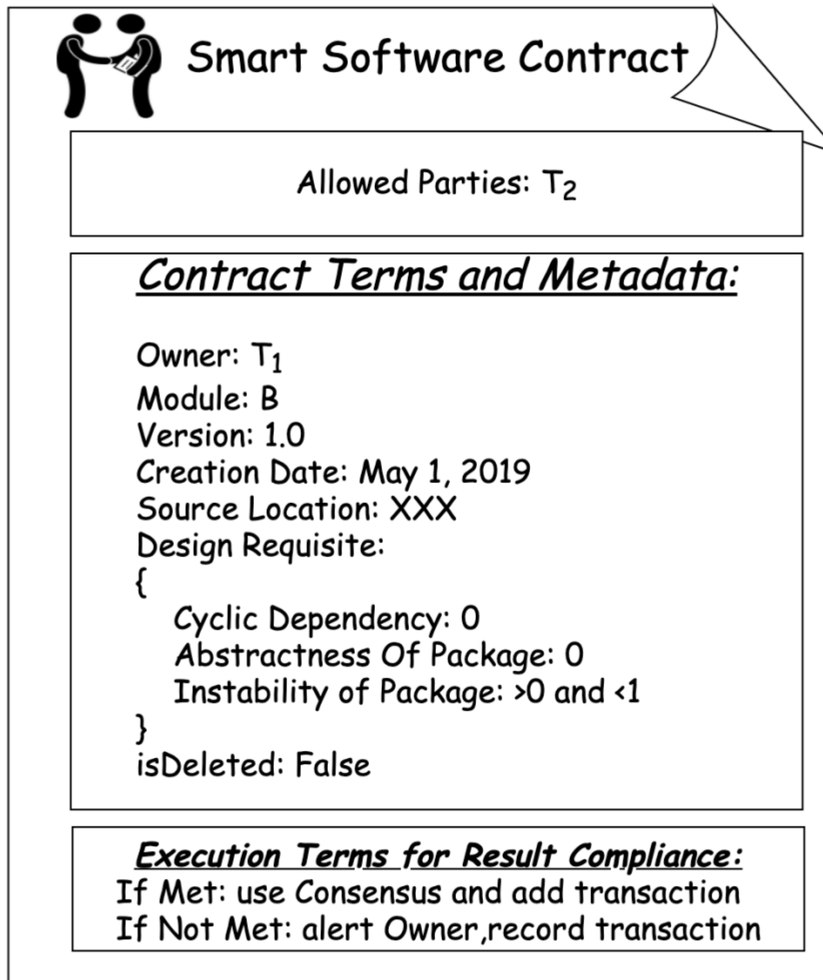


Figure 7: The Smart Contract for Example 1

```
type SoftwareSpec struct {

        AllowedParties Element `json:"allowedParties"`

        Owner  string `json:"owner"`

        Colour string `json:"colour"`

        SoftwareModule string `json:"softwareModule"`

        Version string `json:"version"`

        CreationDate time.Time `json:"creationDate"`

        SourceLocation string `json:"sourceLocation"`
```

```
	designRequisites make(map[string]Element) string `json:"designRequisites"`
	isDeleted bool `json:"isDeleted"`
}
// Define the Smart Contract structure
type SmartContract struct {
}
// init method implementation
func (s *SmartContract) Init(APIstub shim.ChaincodeStubInterface) sc.Response
{
	return shim.Success(nil)
}
```

Above struct is populated with the data for the software design phase at the time of instantiation, which look like follows:

```
type SoftwareSpec struct {
	var allowedParties = [T2] // contains list of allowed teams
	var owner = 'T1'
	var softwareModule = 'CB'
	var version = '1.0'
	date creationDate = 'May 1, 2019'
	var sourceLocation = '/var/sdlc/cse599/projectcoll/classDiagram'
	var designRequisites = { // key-value pair of each acceptance criteria
		var cyclicDependency = [0, T2]
		var abstractnessOfPackage = [0.5, T2]
```

```
                var instabilityOfPackage = [(0,1), T2]
        }
        var isDeleted = False
}
```

Note that, each design requisites are assigned to one of the allowed parties. In case there are multiple parties involved in the same software development phase, each requisite is assigned to the same or a different party.

- Invoke: invoke method is used to provide different actions in a smart contract. Based on the requirement of the software development project, the team can specify unique functionality in each corresponding method. For smart contract mentioned in Figure 7, the following methods are implemented:
    - query: query method implements logic to query specific attributes in the smart contract structure with current value stored in the blockchain. For example, if T_2 needs to know who the owner of the smart contract is, T_2 run query function to know owner information. Similarly, any information regarding the current status of the information is fetched using a query function.

      Implementation of query function in the go programming language is as follows:

```
func (s *SmartContract) query(APIstub shim.ChaincodeStubInterface,
    args []string) sc.Response {
    if len(args) != 1 {
            return shim.Error("Incorrect number of arguments.
                    Expecting 1")
```

```
        }
        dataAsBytes, _ := APIstub.GetState(args[0])
        return shim.Success(dataAsBytes)
    }
```

Sample query calls:

Invoke query owner (returns owner of the smart contract)

Invoke query designRequisites (returns list of design requisites for the software component)

Invoke query souceLocation (returns location of software specification)

- audit: audit method implements logic to pull entire history related to this smart contract. Audit is useful to get an immutable history of the software component regarding all previous software decisions and software results provided by the same or other teams.

  Implementation of audit function in the go programming language is as follows:

```
func (s *SmartContract) audit(APIstub shim.ChaincodeStubInterface)
sc.Response {
  startKey := "SS0"
  endKey := "SS999"
  resultsIterator, err := APIstub.GetStateByRange(startKey, endKey)
  if err != nil {
        return shim.Error(err.Error())
  }
```

```
defer resultsIterator.Close()

// buffer is a JSON array containing QueryResults

var buffer bytes.Buffer

buffer.WriteString("[")

bArrayMemberAlreadyWritten := false

for resultsIterator.HasNext() {

	queryResponse, err := resultsIterator.Next()

	if err != nil {

		return shim.Error(err.Error())

	}

	if bArrayMemberAlreadyWritten == true {

		buffer.WriteString(",")

	}

	buffer.WriteString("{\"Attribute\":")

	buffer.WriteString("\"")

	buffer.WriteString(queryResponse.Key)

	buffer.WriteString("\"")

	buffer.WriteString(", \"Attribute Value\":")

	buffer.WriteString(string(queryResponse.Value))

	buffer.WriteString("}")

	bArrayMemberAlreadyWritten = true

}

buffer.WriteString("]")
```

```
  fmt.Printf("- audit All Software Specification results :\n %s \n", buffer.String())
  return shim.Success(buffer.Bytes())
}
```

- update: update method implements logic to query specific attribute in the smart contract structure with current value stored in the blockchain and then update that attribute with user-supplied value. For example, if T_1 needs to update the owner of the smart contract, T_2 run update function to know owner information and then update it with new user-supplied value. Similarly, any information regarding the current status of the information is updated using the update function.

  Implementation of audit function in the go programming language is as follows:

```
func (s *SmartContract) update(APIstub shim.ChaincodeStubInterface, args []string) sc.Response {
  if len(args) != 2 {
        return shim.Error("Incorrect number of arguments. Expecting 2")
  }
  dataAsBytes, _ := APIstub.GetState(args[0])
  softwareSpec := SoftwareSpec{}
  json.Unmarshal(dataAsBytes, &softwareSpec)
  softwareSpec.Owner = args[1]
  dataAsBytes, _ = json.Marshal(softwareSpec)
```

```
  APIstub.PutState(args[0], dataAsBytes)
  return shim.Success(nil)
}
```

Sample update calls:

Invoke update owner T_3 (update the owner of the smart contract as T_3 in the blockchain)

Invoke update allowedParties [T_2 , T_4] (update list of allowed parties for the software component)

Note that, each update function query generates log and add that into the blockchain. Any team can verify how many times a specific smart contract got updated. Additionally, in the approach, all minor versions are handled by updating existing attributes. The approach suggests that team generates a separate smart contract for the major version change in the software component.

- delete: delete method implements logic to remove specific attributes in the smart contract structure with current value stored in the blockchain or decommission the entire smart contract from the blockchain. If the team deletes any attribute or decommission smart contract, previous data related to smart contact will remain in the blockchain.

  Implementation of delete function in the go programming language is as follows:

```
func (t *SmartContract) delete(stub shim.ChaincodeStubInterface, args
[]string) pb.Response {
  if len(args) != 1 {
```

```
            return shim.Error("Incorrect number of arguments. Expecting 1")
    }
    softwareSpec.Owner := args[0]
    err := stub.DelState(softwareSpec.Owner)
    if err != nil {
            return shim.Error("Failed to delete state")
    }
    return shim.Success(nil)
}
```

Sample query calls:

Invoke delete owner (removes owner attribute from the blockchain structure)

Invoke delete creationDate (removes creationDate attribute from the blockchain structure)

- processResult: processResult method implements logic to verify supplied software results by participating team. Each supplied result is compared with the requisite attribute of the blockchain structure. If the supplied result satisfies acceptance criteria, then the blockchain transaction is generated with a success message and added to the blockchain. If the supplied result does not satisfy acceptance criteria, then the current owner of the smart contract is alerted of the wrong result through email. Blockchain transaction with wrong supplied result is also recorded in the blockchain for future use. The

participating team's reputation can be derived from the correct and wrong result supply.

Implementation of processResult function in the go programming language is as follows:

```
Func (s *SmartContract) processResult(APIstub
shim.ChaincodeStubInterface, args []string) sc.Response {
  if len(args) != 3 {
          return shim.Error("Incorrect number of arguments. Expecting 3")
  }
  dataAsBytes, _ := APIstub.GetState(args[0])
  softwareSpec := SoftwareSpec{}
  json.Unmarshal(dataAsBytes, &softwareSpec)
  var softwareSpecAttribute = args[1]

  var softwareSpecAttributeVal = args[2]
  var designReqs = softwareSpec.designRequisites
  for key, element := range designReqs {
     fmt.Println("Key:", key, "=>", "Element:", element)
     if key == softwareSpecAttribute{
          if element == softwareSpecAttributeVal{
                 APIstub.PutState("Success", "Team has provided software
                 result compliant to acceptance criteria")
          }
```

```
else{
s.alertOwner(APIstub, args)
APIstub.PutState("Failure", "Team has provided software
result not compliant to acceptance criteria with data -
"+softwareSpecAttribute + " - "+ softwareSpecAttributeVal)
}
}
}
return shim.Success(nil)
}
```

- alert: alert method implements logic to send an email message to the owner of the smart contract with a customized message.

5.2 Software Testing Smart Contract

The Software testing phase of software development is a critical phase and requires a major amount of coordination. Software testing consists of the process of verification and validation of the developed software. Based on the criticality of the software project, this phase requires many teams to perform software testing activities on the software. The software testing phase plays a significant role in the success of the collaborative software development project. Software testing can be assessed using several matrices, like line coverage, branch coverage, number of medium and severe bugs. A smart contract for the software testing phase of a collaborative software development project includes the necessary information to perform software testing and acceptance criteria for software

testing evaluation. Participating team decides consensus criteria from pBFT and round-robin based on the requirement of the software component and relation among participating teams. Following the above process Step 1) in Section 5.1 input (a, b, and c) are generated for the smart contract shown in Figure 8.

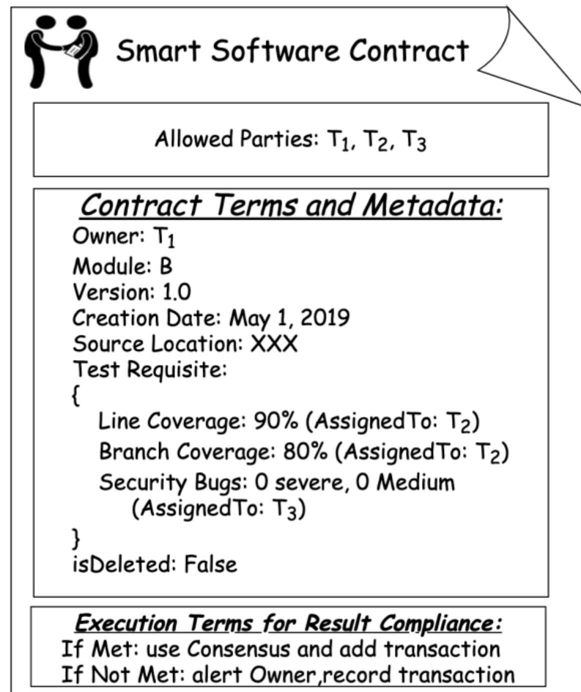


Figure 8: The Smart Contract for Example 2

According to Step 2) in Section 5.1 following methods are generated and implemented to carry out software activity mentioned in the smart contract for software testing phase:

Init: Instantiate structure or class with required software testing metadata, all assessment requirement for a software component can be added into one smart contract. Sample code in go programming language is shown as follows for the smart contract:

```
type SoftwareSpec struct {
    AllowedParties Element `json:"allowedParties"`
```

```
        Owner  string `json:"owner"`
        Colour string `json:"colour"`
        SoftwareModule string `json:"softwareModule"`
        Version string `json:"version"`
        CreationDate time.Time `json:"creationDate"`
        SourceLocation string `json:"sourceLocation"`
        testingRequisites make(map[string]Element) string `json:"testingRequisites"`
        isDeleted bool `json:"isDeleted"`
}
// Define the Smart Contract structure
type SmartContract struct {
}
// init method implementation
func (s *SmartContract) Init(APIstub shim.ChaincodeStubInterface) sc.Response
{
        return shim.Success(nil)
}
```

Above struct is populated with the data for the software testing phase at the time of instantiation, which look like follows:

```
type SoftwareSpec struct {
        var allowedParties = [T1, T2, T3]
        var owner = 'T1'
        var softwareModule = 'CB'
```

```
        var version = '1.0'
        date creationDate = 'May 1, 2019'
        var sourceLocation = '/var/sdlc/cse599/projectcoll/homepage.java'
        var testingRequisites = {
        var lineCoverage = [0.90, T2]
        var branchCoverage = [0.5, T2]
        var securityBugsSevere = [0, T3]
        var securityBugsMedium = [0, T3]
        }
        var isDeleted = False
}
```

Note that, each testing requisites are assigned to one of the allowed parties. In case there are multiple parties involved in the same software development phase, each requisite is assigned to the same or different party.

- Invoke: invoke method is used to provide different actions in a smart contract for the software testing phase. Based on the requirement of the collaborative software development project, the team can specify unique functionality in each corresponding method for software testing. For smart contract mentioned in Figure 8, the following methods are implemented:
    - query: query method implements logic to query specific attributes in the smart contract structure with current value stored in the blockchain. For example, if T_3 needs to know who are allowed parties for this smart contract, T_3 run query function to know allowedParties information. Similarly, any information

regarding the current status of the information is fetched using a query function. Method implementation of the query method is shown in Section 5.1

Sample query calls:

Invoke query allowedParties (returns list of allowed parties for the smart contract)

Invoke query souceLocation (returns location of software specification)

- audit: audit method implements logic to pull entire history related to this smart contract. Audit is useful to get an immutable history of the software component regarding all previous software decisions and software results provided by the same or other teams. Method implementation of the audit method is shown in Section 5.1

  Sample audit calls:

  Invoke audit (return all blockchain transaction related to smart contract)

- update: update method implements logic to query specific attribute in the smart contract structure with current value stored in the blockchain and then update that attribute with user-supplied value. For example, if T_2 needs to update the owner of the smart contract, T_2 run update function to know owner information and then update it with new user-supplied value. Similarly, any information regarding the current status of the information is updated using the update function. Method implementation of the update method is shown in Section 5.1

  Sample update calls:

Invoke update owner T_3 (update the owner of the smart contract as T_3 in the blockchain)

Invoke update isDeleted True (update attribute isDeleted to True and calls blockchain method to decommission entire smart contract)

Note that, each update function query generates log and add that into the blockchain. Any team can verify how many times a specific smart contract got updated.

- delete: delete method implements logic to remove specific attributes in the smart contract structure with current value stored in the blockchain or decommission the entire smart contract from the blockchain. If the team deletes any attribute or decommission smart contract, previous data related to smart contact will remain in the blockchain. Method implementation of the delete method is shown in Section 5.1

  Sample query calls:

  Invoke delete owner (removes owner attribute from the blockchain structure)

  Invoke delete creationDate (removes creationDate attribute from the blockchain structure)

- processResult: processResult method implements logic to verify supplied software results by participating team. Each supplied result is compared with the requisite attribute of the blockchain structure. If the supplied result satisfies acceptance criteria, then the blockchain transaction is generated with a success message and added to the blockchain. If the supplied result does not

satisfy acceptance criteria, then the current owner of the smart contract is alerted of the wrong result through email. Blockchain transaction with wrong supplied result is also recorded in the blockchain for future use. The participating team's reputation can be derived from the correct and wrong result supply. Method implementation of the processResult method is shown in Section 5.1

- alert: alert method implements logic to send an email message to the owner of the smart contract with a customized message.

5.3 Software Requirement Smart Contract

The most difficult phase in any software development project is the requirement phase. It is crucial for software developers to do this phase correctly because all other phases depend on the result of this phase. The requirement phase apprehends user's expectations out of software and showcases what is user's need from the software. Software requirements are supplied in natural language to software development teams, and software development teams convert it to user stories (in the latest paradigms like agile) to breakdown requirements into technical tasks. The encoding of the requirement phase into a smart contract is most difficult for the blockchain-based approach.

The requirement phase is breakdown into several steps to manage in the blockchain-based approach as follows:

- Collect inputs from stakeholders, users, and other software development actors.
- Analyze requirement inputs and convert them into user story format like
  "As a user _____, I want to ______, so that _______."

This format captures three important information on actor, task/action, and motivation behind the software requirement.

- Enumerate all such software requirements into requirement traceability matrix and add required metadata information on each software requirement. [24]

  The blockchain-based approach is capturing the following information for each software requirement:

  - Requirement Id: unique enumeration id for the software requirement
  - Effort: effort required to develop the task mentioned in the requirement.

    Possible values - {Low, Medium, High}
  - Risk: risk associated with the requirement

    Possible values - {Low, Medium, High, Severe}
  - Stability: stability index for the requirement in whole software development

    Possible values - {Low, Medium, High}
  - Status: current status of the requirement in the software development

    Possible values - {Proposed, Ready, In-progress, Ready-for-Review (Validation-and-Verification), Completed, Need-Further-Information}
  - ReleaseDate: {Date or undefined}

A smart contract for the software requirement phase of a collaborative software development project includes the necessary information to perform software requirement elicitation, requirement management, and acceptance criteria for software requirement evaluation. Following the above process, Step 1) generates input (a, b, and c) for the smart contract shown in Figure 9.

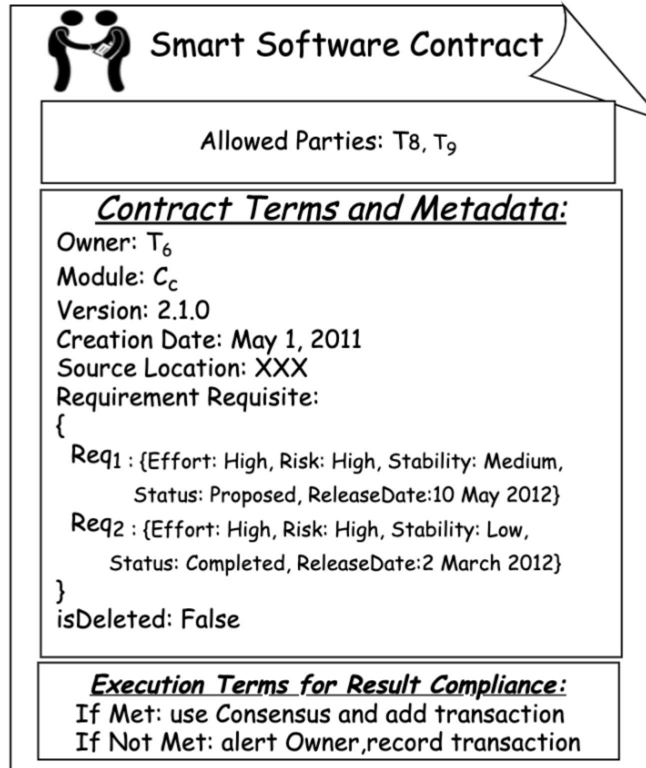


Figure 9: The Smart Contract for Example 3

According to Step 2) of Section 5 following methods generate to carry out software activity mentioned in the smart contract:

Init: Instantiate structure or class with required software specification metadata.

Sample code in go programming language is shown as follows for the smart contract:

```
type SoftwareSpec struct {
    AllowedParties Element `json:"allowedParties"`
    Owner  string `json:"owner"`
    Colour string `json:"colour"`
    SoftwareModule string `json:"softwareModule"`
    Version string `json:"version"`
```

```
        CreationDate time.Time `json:"creationDate"`
        SourceLocation string `json:"sourceLocation"`
        requirementRequisites make(map[string]Element) string
        `json:"requirementRequisites"`
        isDeleted bool `json:"isDeleted"`
}
// Define the Smart Contract structure
type SmartContract struct {
}
// init method implementation
func (s *SmartContract) Init(APIstub shim.ChaincodeStubInterface) sc.Response
{
        return shim.Success(nil)
}
```

Above struct is populated with the data for the software requirement phase at the time of instantiation, which look like follows:

```
type SoftwareSpec struct {
        var allowedParties = [T8, T9]
        var owner = ‘T6’
        var softwareModule = ‘CC’
        var version = ‘2.1.0’
        date creationDate = ‘May 1, 2011’
        var sourceLocation = ‘/var/sdlc/cse599/projectcoll/requirementTraceability.xlsx’
```

var requirementRequisites = {

[ID: Req_1, Effort: High, Risk: High, Stability: Medium, Status: Proposed, ReleaseDate:10 May 2012, assignedTo: T_8],

[ID: Req_2, Effort: High, Risk: High, Stability: Low, Status: Completed, ReleaseDate:2 March 2012, assignedTo: T_9]

}

var isDeleted = False

}

Note that each requirement requisites are assigned to one of the allowed parties. In case there are multiple parties involved in the same software development phase, each requisite is assigned to the same or a different party.

- Invoke: invoke method is used to provide different actions in a smart contract. Based on the requirement of the collaborative software development project, the team can specify unique functionality in each corresponding method. For smart contract mentioned in Figure 9, the following methods are implemented:
  - query: query method implements logic to query specific attributes in the smart contract structure with the current value stored in the blockchain. For example, if T_8 needs to know who are allowed parties for this smart contract, T_8 runs the query function to know the requirement requisites information. Similarly, any information regarding the current status of the information is fetched using the query function. Method implementation of the query method is shown in Section 5.1

    Sample query calls:

Invoke query requirementRequisites (returns list of all requirement requisites in the smart contract)

- audit: audit method implements logic to pull entire history related to this smart contract. Audit is useful to get an immutable history of the software component regarding all previous software requirement decisions and software results provided by the same or other teams. Method implementation of the audit method is shown in Section 5.1

  Sample audit calls:

  Invoke audit (return all blockchain transaction related to requirement smart contract)

- update: update method implements logic to query specific attribute in the smart contract structure with current value stored in the blockchain and then update that attribute with user-supplied value. For example, if T_6 needs to update the owner of the smart contract, T_6 run update function to know owner information and then update it with new user-supplied value. Similarly, any information regarding the current status of the information is updated using the update function. Method implementation of the update method is shown in Section 5.1

  Sample update calls:

  Invoke update isDeleted True (update attribute isDeleted to True and calls blockchain method to decommission entire smart contract)

Note that, each update function query generates log and add that into the blockchain. Any team can verify how many times a specific smart contract got updated.

- delete: delete method implements logic to remove specific attributes in the smart contract structure with the current value stored in the blockchain or decommission the entire smart contract from the blockchain. If the team deletes any attribute or decommission smart contract, previous data related to smart contact will remain in the blockchain. Method implementation of the delete method is shown in Section 5.1

  Sample query calls:

  Invoke delete owner (removes owner attribute from the blockchain structure)

  Invoke delete creationDate (removes creationDate attribute from the blockchain structure)

- processResult: processResult method implements logic to verify supplied software results by participating team. Each supplied result is compared with the requisite attribute of the blockchain structure. If the supplied result satisfies acceptance criteria, then the blockchain transaction is generated with a success message and added to the blockchain. If the supplied result does not satisfy acceptance criteria, then the current owner of the smart contract is alerted of the wrong result through email. Blockchain transaction with wrong supplied result is also recorded in the blockchain for future use. The participating team's reputation can be derived from the correct and wrong

result supply. Method implementation of the processResult method is shown in Section 5.1

- alert: alert method implements logic to send an email message to the owner of the smart contract with a customized message.

CHAPTER 6

ILLUSTRATIVE EXAMPLE OF THE APPROACH

In this section, an example to illustrate the overall approach is presented to generate trusted coordination of a collaborative software development project.

Consider the collaborative software development project, which has multiple major software components developed by multiple teams. In this example, the assumption in the approach is that the software system to be collaboratively developed has six major software components, C_a, C_b, … C_f, collaboratively developed by three teams T_1, T_2, and T_3. The development of each software component is complex and requires input from multiple teams. Role assignment for each software component and various phases of software development phases are assigned at the time of blockchain initialization. Each software component is developed in various software development phases, such as requirement gathering and analysis, software design, software implementation, software testing, deployment, and maintenance.

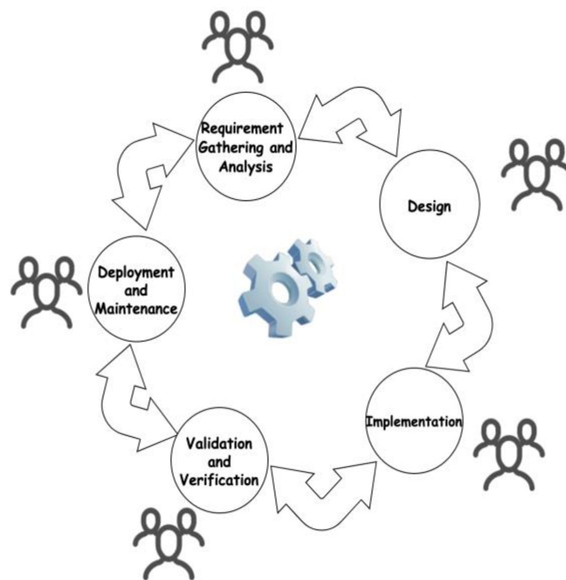


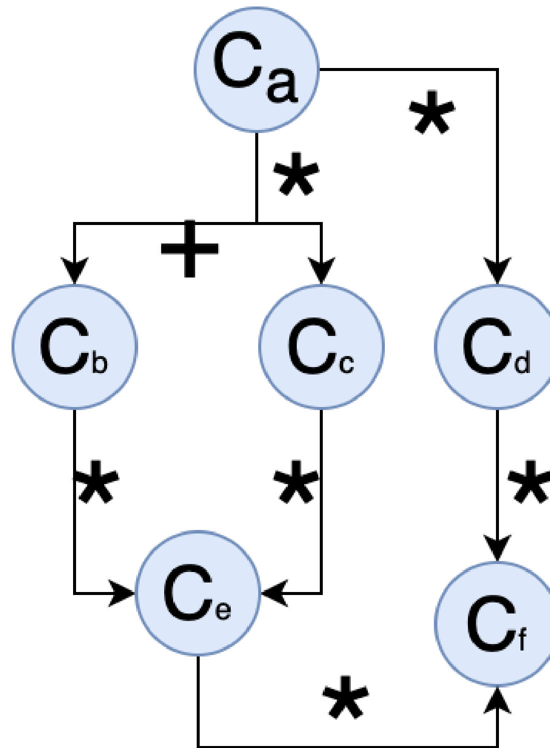
Figure 10: The Software Development Activities Required for Each Software Component

Software components used in the example are complex enough to have different teams participating in the collaborative software development. As shown in Figure 10, each of the six components will have various software development phases, and each phase is handled by one or more teams. This method is suitable for all kinds of software development paradigms, such as waterfall, spiral, agile. Coordination is an essential part of all kinds of collaborative software development.

The functional dependency relations among the six components, which are determined by the requirements of the software development project, are shown in Figure 11. Each dependency relation in Figure 11 can be either mandatory or optional. Dependency information is essential information in complex and large-scale software development projects to understand the requirement of other components as well as to assess the reliability of frequently used software components. As shown in Figure 11, Component C_a is an end software product. To develop software component C_a, component C_d is mandatory, and either of C_b or C_c is required. Additionally, Component C_e is mandatory to develop component C_b and C_c. C_f is a mandatory component to develop C_e and C_d. Trusted coordination is required to build such a complex and large-scale software project.

It is assumed that T_1 is the prime contractor, and T_2 and T_3 are subcontractors in the collaborative software development project. Based on the discussion of T_1 with T_2 and T_3, the development tasks of this project will be performed by the three teams according to the software specifications listed in Table 2. T_1, T_2, and T_3 manage all software development activities in a collaborative project. This example shows how the approach

can generate trusted coordination among the three teams using blockchain and smart contracts.



* represents mandatory, and + represents optional dependency relation

Figure 11: The Functional Dependency Graph of the Major Software Components in the Example Project.

The coordination in a collaborative software development project consists of collaboration among multiple teams working for developing various software components in a complex and large-scale software project. Each software component, including the software specifications and results of the component, is stored in the blockchain. In this example, the only illustration of the coordination among the teams participating in the software development of C_f is shown. The same method can be extended for all remaining software components.

According to Step 1 of the overall approach described in Chapter 4, T_1 creates software specification for the software component C_f, and smart contract $C_{1,k}$, $T_k \in G_1$, $G_1 = \{T_2, T_3\}$. $C_{1,k}$ is generated using the process presented in Chapter 5. T_2 and T_3 use $C_{1,k}$ generated in Step 1 to carry out software activities for the C_f. According to Step 2 of the approach, T_2 performs the software activity specified in $C_{1,2}$ and generates the result R_1. T_3 performs the software activity specified in $C_{1,3}$, and generates the result R_2.

Validation of this step to achieve trusted coordination:

- R_1 and R_2 are recorded in the blockchain, and hence any team in the blockchain can verify them with known public keys of participating teams.
- Once R_1 or R_2 is recorded in the blockchain, it is challenging to modify them; this provides tamper-evidence and tamper-resistant tool for coordination.
- T_2 and T_3 cannot repudiate R_1 or R_2 because their identities are attached to transactions and available to verify in the blockchain.
- T_1, T_2, and T_3 cannot take control of the software development activities, due to decentralized and distributed properties of the approach.
- T_1 can clearly set up contract terms for T_2 and T_3 for collaboration, this will help to resolve any conflicts among teams, and teams need to adhere to acceptance criteria for created software development.

Since the procedure, as mentioned above, applies to all software components in all software development phases, the entire history of the component is available in the blockchain in terms of smart contracts. Transaction R_1 and R_2 are added in the blockchain if it is meeting consensus criteria. The approach uses the Practical Byzantine Fault Tolerance consensus model to add the new next block. This consensus collects voting from all participants, and

if teams provide one-third of the agreement, the block will be added in the blockchain. The approach used round-robin consensus in a few smart contracts, but pBFT is more useful to establish trust among participating teams.

Software Component	*Versions*	*Various Software Phases assigned to Team/s*	*Private Communication required?*
Ca	1.0, 1.1, 2.0	Requirement: T1 Design: T2 Implementation: T2 Testing: T2 Maintenance: T2	No
C_b	1.0	Requirement: T2 Design: T2 Implementation: T1,T3 Testing: T2 Maintenance: T2	No
Cc	1.0, 2.0, 3.0	Requirement: T1, T2 Design: T2, T3 Implementation: T1 Testing: T2 Maintenance: T2	No
Cd	1.0, 1.1, 1.2, 1.3	Requirement: T1 Design: T2, T3 Implementation: T1 Testing: T3 Maintenance: T1	No
Ce	1.0	Requirement: T1 Design: T2 Implementation: T1 Testing: T2 Maintenance: T1	Yes
Cf	1.0, 1.1, 2.0, 2.1, 2.2	Requirement: T1 Design: T2, T3 Implementation: T1 Testing: T2,T3 Maintenance: T1	No

Table 2: The Software Component Specifications of the Example Project

Next, the auditability feature for software specifications and results in the project is shown. Because of the dependency relation between C_d and C_f, as shown in Figure 11, in order to develop C_d, T_3 needs to verify the history of C_f for all software decisions taken and related history for different software development phases. According to Step 3, T_3

verifies the history of C_f by querying smart contracts $C_{1,k}$. The querying smart contract is performed by calling one of the smart contract methods discussed in Chapter 5. Since the complete record of the past development history of C_f is stored in the blockchain, any team in the blockchain can verify the entire history of C_f, which is useful to increase the trustworthiness of the coordination in collaborative software development.

Private communication channels:

Private channels are created to share the result of the critical project components. Private channels are established using asymmetric key pairs if teams want to establish private communication. They can share their public keys without sharing it with anyone else. Each team can encrypt software results with their private key, and other parties can see the test result by decrypting result using the public key of the transaction initiator.

The blockchain-based approach is implemented on a Hyperledger platform [23], which is a private/permissioned blockchain platform. The reason why the approach is implemented on the Hyperledger platform, as mentioned below:

- The collaborative software development project is not open to the public, only approved and pre-verified teams can join the blockchain platform, so the selection of private blockchain is ideal for this approach. All participating team is known and pre-verified, this increases partial trust among software development teams and eliminates the anonymity feature of most blockchain platforms.
- As more teams join the software development activity, blockchain should scale accordingly. Private blockchain scales easily as compared to the known public blockchain platform.

- Transaction processing is faster in private blockchain set up due to the absence of a resource-intensive mining process and proof of work consensus model. Private blockchain, like Hyperledger, provides low latency and high throughput for the blockchain transaction.
- The various consensus is utilized and pluggable based on the requirement of the team.
- Teams set up a private communication channel to share software data on a critical software project.
- The team leverages the reputation system for producing high-quality collaborative software.

CHAPTER 7

EVALUATION

In this chapter, we will compare our approach to the existing approaches in the categories using centralized, [2-10] or decentralized [7][12] approaches to generating coordination in collaborative software development for complex and large-scale software systems.

The centralized approaches [2-10] have the disadvantages of a single point of failure, lack of data ownership, and mutable data. The centralized approaches have the advantages of providing much faster data processing speed, and higher efficiency of the setup of the distributed environment for collaborative software development.

The decentralized approaches [7][12] have the limitations on lack of data ownership, mutable data, and concentrated point of failure. The decentralized approaches have moderate data processing speed, and moderate efficiency of the setup of the collaborative environment for collaborative software development.

Our approach using blockchain has limitations on low data processing speed, large storage overhead, and lower efficiency of the setup of the collaborative environment. The low data processing speed is due to the consensus protocols used for recording each transaction in the blockchain for data ownership, auditability, and the prevention of injection attacks in the software development process. The large storage overhead is due to many copies of information stored in the blockchain. The lower efficiency of the setup of the collaborative environment is due to the initial blockchain infrastructure setup, and cryptographic tools installation in the blockchain.

Our approach has the major advantages of the immutability, auditability, non-repudiation, and acceptance criteria for automatic software assessment for generating trusted coordination for collaborative software development.

Our approach has the advantage of immutability of data in blockchain due to cryptographic links and Merkle root hash in the blockchain. The advantage of auditability of all blockchain transactions in our approach is due to the smart contract created to query all the blockchain transactions. Our approach has the advantage of non-repudiation because each blockchain transaction is cryptographically signed with its node's identity. Our approach has the advantage of having acceptance criteria for automatic software assessment because the software specifications are encoded in each smart contract with software data and acceptance criteria.

No	Properties	Centralized Approach	Decentralized Approach	Our Approach
i	Immutability	No	No	Yes
ii	Auditability	No	No	Yes
iii	Consensus	No	No	Yes
iv	Non-repudiation	No	No	Yes
v	Private communication	No	No	Yes
vi	Storage Overhead	Low	Moderate	High
vii	Data Processing Speed	High	Moderate	Low
viii	Setup Efficiency	High	Moderate	Low

Table 3: A Summary of the Important Properties of Our Approach, Centralized, and Decentralized Approaches for Generating Trusted Coordination of Collaborative Software Development

Table 3 summarizes the important properties of our approach, centralized and decentralized approaches to generating trusted coordination of collaborative software development. The first four properties are needed to generate trusted coordination of collaborative software development because they are required to provide a secure, reliable, and transparent collaborative environment. The 5th property is optional because a private communication is required for critical software development activities only. The 6th, 7th and 8th properties are limitations of using blockchain in providing trusted coordination of collaborative software development. The 6th property can be reduced by storing partial data off-chain. The 7th property can be reduced by using various custom consensus protocols for the private blockchain. The impact of the 8th property is not large because it only affects the initialization phase for setting up the blockchain infrastructure. Based on the above discussion, our approach can achieve the generation of trusted coordination of collaborative software development because it has the first four properties, the impact of the 8th property is limited to the initialization phase, and the 6th and 7th properties may be reduced.

CHAPTER 8

CONCLUSION AND FUTURE WORK

In this thesis, the generation of trusted coordination in collaborative software development using blockchain has been studied, and an approach to generating trusted coordination of collaborative software development has been developed. This approach has been implemented using the private blockchain in open-source Hyperledger. The advantages of our approaches are immutability, auditability, non-repudiation, and acceptance criteria for automatic software assessment. The limitations of our approach are large storage overhead, low speed of data processing, and large setup overhead.

Future work includes the development of efficient techniques to reduce the overhead of the data storage in blockchain and increase the speed of data processing. In addition, the approach should be expanded to generate trusted coordination of multiple collaborative software development projects.

REFERENCES

[1] Schmidt, K., and Bannon, L. (1992). Taking CSCW seriously. Computer Supported Cooperative Work (CSCW), 1(1), 7-40.

[2] Hefner, R. H. (2001). Collaborative test management. 26th Annual NASA Goddard Software Engineering Workshop, IEEE/NASA SEW 2001, 156–158.

[3] Zhu, H., & Zhang, Y. (2012). Collaborative testing of web services. IEEE Transactions on Services Computing, 5(1), 116–130.

[4] Kukreja, S., Singhal, A., & Bansal, A. (2015). A critical survey on test management in IT projects. International Conference on Computing, Communication and Automation, ICCCA 2015, 791–796.

[5] Bogner, J., Boceck, T., Popp, M., Tschechlov, D., Wagner, S., & Zimmermann, A. (2019). Towards a Collaborative Repository for the Documentation of Service-Based Antipatterns and Bad Smells. In Proceedings - 2019 IEEE International Conference on Software Architecture - Companion, ICSA-C 2019 (pp. 95–101). Institute of Electrical and Electronics Engineers Inc.

[6] https://taiga.io/

[7] https://confluence.atlassian.com/

[8] Park, J. S., Suresh, A. T., An, G., & Giordano, J. (2006). A framework of multiple-aspect component-testing for trusted collaboration in mission-critical systems. In 2006 International Conference on Collaborative Computing: Networking, Applications and Worksharing, CollaborateCom.

[9] Jhala, K. S., Oak, R., & Khare, M. (2018). Smart collaboration mechanism using blockchain technology. Proceedings - 5th IEEE International Conference on Cyber Security and Cloud Computing and 4th IEEE International Conference on Edge Computing and Scalable Cloud, CSCloud/EdgeCom 2018, 117–121.

[10] Zheng, Zibin, Shaoan Xie, Hongning Dai, Xiangping Chen, and Huaimin Wang. "An overview of blockchain technology: Architecture, consensus, and future trends." In 2017 IEEE international congress on big data (BigData congress), pp. 557-564. IEEE, 2017.

[11] https://www.idtheftcenter.org/data-breaches/

[12] "Initial revision of "git", the information manager from hell". GitHub. 8 April 2005.

[13] Nakamoto S. (2008). Bitcoin: A peer-to-peer electronic cash system, Available at http://bitcoin.org/bitcoin.pdf

[14] Yaga, Dylan, Peter Mell, Nik Roby, and Karen Scarfone. "Blockchain technology overview." arXiv preprint arXiv:1906.11078 (2019).

[15] Becker, Georg (2008-07-18). "Merkle Signature Schemes, Merkle Trees and Their Cryptanalysis" (PDF). Ruhr-Universität Bochum. p. 16. Retrieved 2013-11-20.

[16] H. Sukhwani, J. M. Martínez, X. Chang, K. S. Trivedi and A. Rindos, "Performance Modeling of PBFT Consensus Process for Permissioned Blockchain Network (Hyperledger Fabric)," 2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS), Hong Kong, 2017, pp. 253-255.

[17] Szabo, N. “Smart Contracts,” 1994.

[18] Szabo N. (1997). Formalizing and securing relationships on public networks. First Monday 2.9

[19] Long, T., Yoon, I., Memon, A., Porter, A., & Sussman, A. (2014). Enabling collaborative testing across shared software components. CBSE 2014 - Proceedings of the 17th International ACM SIGSOFT Symposium on Component-Based Software Engineering (Part of CompArch 2014), 55–64.

[20] Long, T., Yoon, I., Porter, A., Memon, A., & Sussman, A. (2016). Coordinated Collaborative Testing of Shared Software Components. Proceedings - 2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016, 364–374.

[21] Wanderley, G. M. P., Abel, M. H., Barthes, J. P., & Paraiso, E. C. (2017). An advanced collaborative environment for software development. 2016 IEEE International Conference on Systems, Man, and Cybernetics, SMC 2016 - Conference Proceedings, 2917–2922.

[22] Scott, B., Loonam, J., & Kumar, V. (2017). Exploring the rise of blockchain technology: Towards distributed collaborative organizations. Strategic Change, 26(5), 423–428.

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

[24] M. Lang & J. Duggan, “A Tool to Support Collaborative Software Requirements Management” Requirements Engineering Journal 6(3), 2001, pp. 161–172.

BIOGRAPHICAL SKETCH

Jinal S. Patel is a graduate student in Software Engineering at Arizona State University. She graduated with a BS degree from L. D. College of Engineering from Gujarat, India, in 2015. She is currently a research assistant at the Center for Cybersecurity and Digital Forensics at ASU. She was teaching assistant for the course software security in the Spring 2020 semester at ASU. During her graduate program, she worked as an application security intern at Silicon Valley bank in Summer 2019 and worked as a Software development Engineer in Test at EdPlus from November 2018 to November 2019. She has worked as Security Engineer at Wipro from 2015 to 2018. Her research interests include software engineering, application security, blockchain, and cybersecurity. She has accepted a full-time offer of Security Engineer at Amazon in the Information Security group at Amazon HQ, Seattle.