PCS: Privacy Preserved Crowdsourcing Services Implemented on a Notary-based Blockchain System

Qiancheng Fan

4th Year Project Report
Artificial Intelligence and Computer Science
School of Informatics
University of Edinburgh

2023

Abstract

This project implements PCS, a privacy-preserved crowdsourcing system implemented on a notary-based permissioned DLT platform known as ”Corda”. The paper discusses the limitations of traditional crowdsourcing systems, such as the single point of failure problem, privacy breaches, and non-transparent assessment, and how PCS addresses these issues using a decentralised blockchain platform and smart contract-based approach to service delivery. Moreover, the paper discusses the limitations of previous approaches aimed at enhancing traditional crowdsourcing systems and outlines how this project differs from them.

PCS provides robust protection for participants' privacy by using the private blockchain Corda for service processing and Paillier homomorphic encryption for task results. The PCS system was designed with a hierarchical structure, and developers can customise it according to their requirements with minimal code modifications. The paper includes a detailed description of the PCS implementation code and comprehensive experiments that evaluate its operational performance.

In conclusion, the paper demonstrates that PCS provides a more secure and transparent approach to crowdsourcing, successfully addressing the limitations of traditional systems. The paper highlights the advantages of PCS, including its decentralised blockchain platform, smart contract-based approach, and privacy-preserving mechanisms, and suggests potential future work to further improve its stability and usability.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Qiancheng Fan*)

Acknowledgements

First of all, I would like to thank my supervisor, Dr Xiao Chen for his patience in answering my various questions about the project. His advice on the direction of my project and his guidance on paper writing helps me a lot.

And I also need to thank Dr Haiqin Wu for her help in providing suggestions about privacy protection for this project.

Finally, I need to thank my family's continued support for my study.

Table of Contents

Chapter 1

Introduction

1.1 Crowdsourcing System Scenarios

The term ”crowdsourcing” was initially introduced by Jeff Howe in his 2006 paper [24]. In the same year, Howe provided a formal definition of the concept in his blog post [4], in which he described it as the act of outsourcing the work that traditionally done by designated agents to uncertain groups in the form of an open call. Since its inception, the field of crowdsourcing has made significant strides over nearly two decades of research on crowdsourcing systems. Crowdsourcing technology has proven successful in various sectors, including medicine [40], education [28], and commerce [39].

There are many successful applications of crowdsourcing. One prominent example is Wikipedia [41], which has achieved the world's largest encyclopedia site with a small team size by using crowdsourcing. Its 15 million articles are mainly crowdsourced from volunteers around the world, and almost all of which can be edited by anyone with access to the site. Another notable example is Airbnb [33], which provides short-term accommodation services to tourists. Its properties are primarily provided by individual contributors who are willing to rent out their properties. Airbnb's use of crowdsourcing to source properties has allowed it to expand its services to most countries around the world.

A traditional crowdsourcing system, commonly known as a centralized crowdsourcing system, typically involves three distinct groups of actors: requesters, workers and a centralized crowdsourcing platform [27]. In this system, the central platform typically manages the majority of data storage and transmission. As illustrated in Fig.1.1, the users send their requests to the platform, and workers receive tasks from the platform and submit their completed work back to it. The platform evaluates the final reward given to the worker based on the results submitted by the worker and feedback from the user. Furthermore, in the event of a dispute between the requestor and the worker regarding the results, the central platform is responsible for making the final decision.

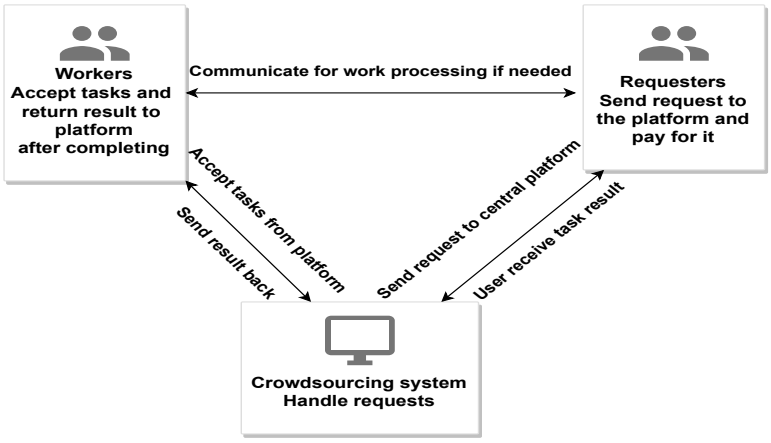


Figure 1.1: System model for traditional crowdsourcing system

1.2 Issues and Challenges

However, the operation of traditional crowdsourcing systems relies heavily on the involvement of a trusted third party (central platform), which presents the following inevitable challenges.

- *Single point of failure problem* [21]. Since traditional crowdsourcing systems relies on a single, centralized platform or server to coordinate and manage tasks, a failure in this platform could disrupt the entire system. This could be due to technical issues, cyberattacks, employee's work failures, force majeure (e.g. earthquake, fire) or other factors that affect the platform's availability or functionality. For example, in 2015, service on the uber system was disrupted due to a hardware failure in Uber China, resulting in passengers in some area unable use Uber services at that time [11].
- *DDos attack*. Just like many systems with a central server, traditional crowdsourcing systems are vulnerable to DDoS attacks, which can cause significant damage. Such attacks are not uncommon. For instance, in March 2014, Elance and oDesk were hit by a DDoS attack that caused service disruptions for numerous freelancers [5].
- *Privacy breach issues*. The central platform of traditional crowdsourcing systems typically stores a significant amount of sensitive information about its participants. They may collect personal information from users, such as their name, delivery address, email address, phone number, and other sensitive data. Furthermore, a user's record of using the platform may reveal additional information about them, such as browsing and purchase history that could reveal their current needs [35], or rental history that could indicate their occupation (for instance, if a user frequently rents Airbnb listings during winter and summer breaks, they are likely a student or teacher). In the event of a security breach on the central platform, data relating to all participants would be available to the attacker, resulting in severe consequences [36][9]. Moreover, in addition to data breaches occurring on the primary platform, crowdsourcing systems require the participation of a

large number of contributors to operate, thereby rendering them susceptible to attacks. Attackers can leverage this vulnerability to carry out malicious activities. For instance, contributors with authorised access to sensitive information may intentionally divulge confidential data or exploit it for their personal gain.

- *Non-transparent assessment*. In traditional crowdsourcing systems, When workers or requesters dispute the outcome, it is usually up to the crowdsourcing platform to make a judgement. Although these platforms have their own set of rules for it, but as this is usually a judgement call by platform staff with invisible decision-making process, the assessment is still subjective and non-transparent. Which makes ”false-reporting”[45] possible.

Previous studies have proposed several solutions to the aforementioned problems, which address them partially. However, none of these solutions are comprehensive enough to resolve all the issues. The related work chapter thoroughly analyzes these solutions and highlights their differences from the proposed solution.

1.3 PCS Solution and Contribution Highlight

To address the aforementioned issues, this project developed a privacy-preserved crowdsourcing service called ”PCS”, which is implemented on a notary-based permissioned Distributed Ledger Technology (DLT) platform known as ”Corda” [16]. In this section, I will describe how this solution solves each of the aforementioned problems.

By utilizing a decentralised blockchain platform, we can distribute the processing and storage of data across a network of nodes, rather than relying on a trusted third party to coordinate and manage tasks [15]. This approach ensures that data is stored in multiple copies across the network of nodes and the failure of a single node does not result in data loss or impact the operation of other nodes. Thus, the network can continue to function and maintain a complete copy of the ledger, which greatly reduces the impact of a single point of failure.

Since a decentralised network does not have a central server, attackers cannot directly target a central server to launch a cyber attack as there is no centralised system to disrupt. Furthermore, this system requires the requester to pay for the request in advance, preventing DDoS attacks by attackers who send a large number of requests but do not pay for them, taking up a large amount of request processing resources. And only registered workers can receive requests, which prevents external nodes from attacking by accepting every task but not executing them. More details about DDoS prevention will be provided later in the security analysis section.

Besides that, as Corda is a permissioned blockchain, we can specify that a particular transaction will only be visible to a selected group of people, in this way, we can let each participant see only the data necessary for them to carry out their own tasks. And for stored data, Corda has its own set of data protection methods, the details of which are mentioned later in the description section of Corda. In addition, I have used Paillier homomorphic encryption[34] in my implementation to calculate contribution rewards. This homomorphic encryption method ensures that some private data can not even be

seen by selected people except the sender themselves while the calculation of the reward can proceed as usual.

The entire crowdsourcing process in this system is implemented through Corda's smart contracts and workflows. Since the content of the smart contract and workflow is public, each step of the process in this solution is open and transparent. Additionally, the process is fully automated, and the determination of the results is not influenced by any personal subjective factors.

In conclusion, my contribution has the following highlight:

- PCS implements crowdsourcing services on a novel permissioned blockchain: Corda.
- PCS's development is based on a hierarchical crowdsourcing model that enables tasks to be effectively divided and delegated among different levels of participants.
- PCS has privacy preserving mechanism to protect participants' sensitive data.
- The current implementation of PCS follows a simulated real-world working scenario, and can be easily modified to suit the requirements of other actual scenario.

The roadmap of this paper is outlined as follows: Chapter 2 reviews the relevant literature in the field of crowdsourcing, analyzing the distinctions between this work and prior studies. In Chapter 3, the fundamental technical background knowledge that is pertinent to this project is presented. Chapter 4 offers a comprehensive description of the system design, outlining the entire system's workflows, the interactions between various participants, and an assessment of the system's privacy and security measures. Chapters 5 and 6 provide details on the system's code implementation and performance evaluation experiments and results analysis, respectively. The final two chapters address the system's achievements and limitations, followed by concluding remarks and potential future research.

Chapter 2

Related Work

This chapter is focused on related works in the field of crowdsourcing, with description of essential background knowledge, and how they differ from the current project. Here, the relevant works in three primary areas that are most relevant to this study will be presented.

2.1 Centralized crowdsourcing system with Anonymity

Privacy issues in crowdsourcing have become a growing concern and have attracted more attention and research in recent years. However, despite this increased attention, these issues are still not extensively studied [43]. Some research suggests that rendering users anonymous might eliminate privacy issues [26]. However, real-world events have shown that even when anonymity is used, there are still possible issues. For instance, in a competition for recommendations system held by Netflix, they provided entrants with datasets of their 480,000 customers. Using this information, the participants successfully determined some users' information are in these datasets [43] [7]. Which means, if a data breach happens, even if the participants in the process are anonymous, an attacker may still use these data to infer private information about the participants.

2.2 Distributed Crowdsourcing System

Distributed crowdsourcing systems are also a popular research topic. In this type of system, they still have a centralised system to handle data and process requests. However, unlike traditional centralised systems, they provide a way to complete task through distributed processing, allowing for better efficiency. For instance, the research [44] proposed an efficient task offloading algorithm based on social relations to improve the task allocation scheme in traditional crowdsourcing systems. Another research [18] designed a Bayesian asynchronous task selection algorithm which is an asynchronous and distributed approach that allows users to use incomplete information about task popularity to make task selections in mobile crowdsourcing.

These research works do increase the crowdsourcing system's efficiency, but using a

distributed approach for task processing does not change the fact that the whole system is still centralised, they still rely on a central platform to make the system work. So the above issues about centralized systems are still not solved.

2.3 Blockchain-based Crowdsourcing System

Improving traditional crowdsourcing systems through blockchain technology is currently the most popular research topic about crowdsourcing. PCS is also a blockchain-based crowdsourcing system. But before I go into specifics of related works, let me give a simple description of what blockchain is and how it is being used in crowdsourcing so that readers can better understand what follows. Blockchain is a decentralised ledger technology that was first used in Bitcoin [32]. In blockchain, records of time-ordered transactions are recorded in a sequence of ”blocks”. Each block contains a hash value of the previous block that allows them to eventually form a public and immutable hash chain of blocks. A typical blockchain-based crowdsourcing system consists of three participants: Requesters, Workers and blockchain. In this system, all the task processing is handled by the blockchain, in specific, they are driven by smart contracts deployed in the blockchain. The idea of smart contract is proposed by Nick Szabo in 1994 [38], they are event-driven programs that have features of self-verifying, self-executing and tamper resistant, and then can use the blockchain's consensus protocol to run a sequence of events [30]. Figure 2.1 is a smart contract basic executing structure diagram from [30]. Because of these properties, a blockchain-based crowdsourcing system can

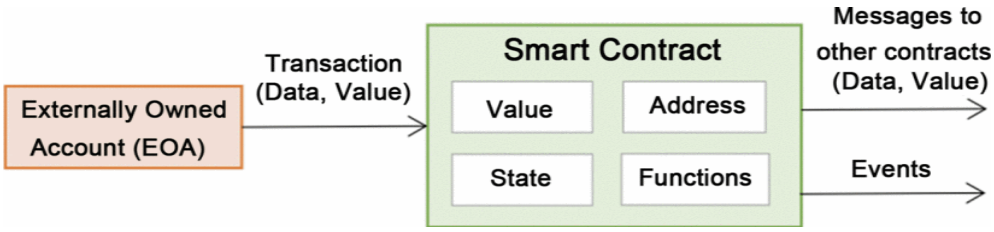


Figure 2.1: A basic structure of smart contract [30]

execute crowdsourcing services without a centralized platform, which effectively solves many of the problems previously mentioned. The decentralized processing approach eliminates the single point of failure problem, and by using smart contract for task handling, since smart contract's content is public and unmodifiable, everyone knows how the assessment will be processed, the issue of non-transparent assessment also does not exist anymore. By adopting this approach, the main issues left to address are data privacy concerns and the risk of denial-of-service attacks.

Then let us discuss some specific works related to blockchain-based solutions in crowdsourcing. The research [27] proposed a crowdsourcing system based on Ethereum [42](a public blockchain platform). Their implementations mainly make use of the features of the public blockchain and apply user anonymity to ensure data privacy. But this method has some potential issues, the attackers can take advantage of the feature of public blockchain to apply attacks. Since all the transactions in public blockchain are shown to the public, although the users involved in the transaction logs are anonymous,

an attacker would still be able to deduce the private information of the participants from the information contained in the bulk of the transaction logs [7]. Another research [14] proposed a blockchain-based crowdsourcing system for court judgement, however, they do not provide a specific code implementation about this system. [25] proposed a Ethereum based crowdsourcing system with incentives built in, but just like [27], it also has similar privacy concerns.

Overall, most of the current work on blockchain-based crowdsourcing systems revolves around famous public blockchains like Ethereum, and not much work has been done around permissioned blockchains, which gives us a motivation to build a crowdsourcing system based on a private(or permissioned) blockchain.

Chapter 3

Technological Background

The goal of this project is to implement a hierarchical crowdsourcing system based on Corda blockchain with privacy protection. In order to allow readers to better understand the work of this project, this chapter introduces some essential background concepts relevant to this project. Section 3.1 introduces the concept of paillier homomorphic encryption, which is an important encryption method used in this project for privacy protection. Section 3.2 describes the platform for realising this project, including its features and the reasons for choosing to use it for this project.

3.1 Paillier Homomorphic Encryption

In this project, I use Paillier cryptosystem's homomorphic property for the assessment of task completion to provide extra privacy protection. The details of the use of this encryption will be described later, this section will focus on the description of Paillier cryptosystem.

The Paillier key system is an additive homomorphic cryptosystem invented by Pascal Paillier in 1999 [34]. This scheme consists of three main algorithms: key generation, encryption and decryption.

In key generation, we have these steps:

1. Generate 2 different random large prime number p and q that have the same length to make sure $\gcd(pq,(p-1)(q-1))=1$.
2. Compute value $n=pq$ and $\lambda=\text{lcm}(p-1,q-1)$.
3. Select a random integer g where $g\in Z^*_{n^2}$.
4. By checking $\mu=(L(g\lambda \bmod n^2))^{-1} \bmod n$, we can ensure n divides the order of g. And the L here is defined as $L(u)=\frac{u-1}{n}$.
5. Now, we can generate the key pairs by using above numbers, where public key is: (n,q), private key is (λ,μ).

Encryption algorithm contains these steps:

1. Let m as the message that we want to encrypt, and $0 < m < n$.
2. Select a random r that $r \in Z_n^*$ and have the property that $\gcd(r,n) = 1$.
3. Then we can generate the ciphertext by calculating $c = g^m r^n \bmod n^2$.

In the decryption algorithm, we need to:

1. Using the generated ciphertext c where $c \in Z_n^*$ and compute the original message by calculating $m = L(c^\lambda \bmod n^2) \cdot \mu \bmod n$.

And according to the explanation from [34], if we have 2 ciphertext c_1 and c_2, the product of them will be $c_1 * c_2 = E(m_1) * E(m_2) = g^{m_1} r_1^n . g^{m_2} r_2^n \ mod \ n^2$, by transformation of equation, we can get $c_1 * c_2 = g^{m_1+m_2} . r_1^n r_2^n mod \ n^2$. Which means the product of ciphertext generated by our encryption algorithm is equal to the encryption of the sum of the two original data. Hence shows the Additive Homomorphic Property of Paillier cryptosystem.

This cryptosystem has already been used in many works, for instance, this research [13] implemented a voting system that guarantees data confidentiality using Paillier homomorphic encryption. Another research by Mahdi Ghadamyari and Saeed Samet [22] designed a privacy-preserved health data analysis system using the Paillier cryptosystem.

3.2 Corda Blockchain

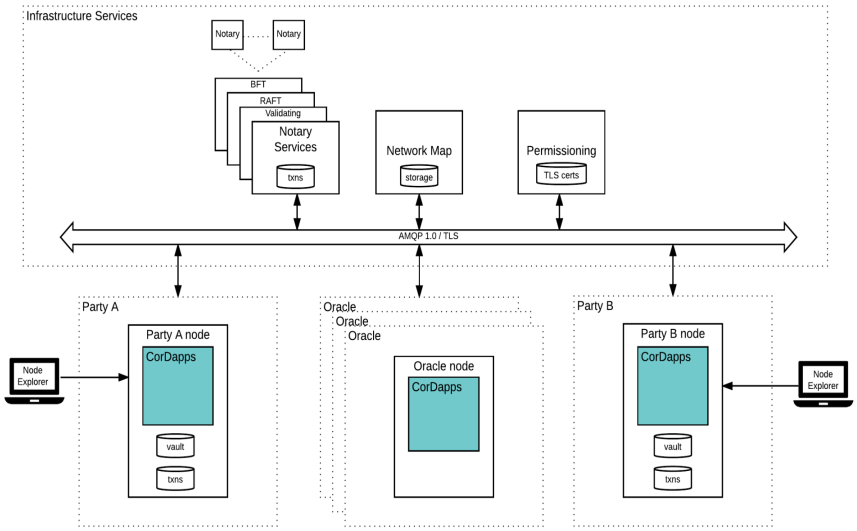


Figure 3.1: Highlevel architecture of Corda [1]

Corda is an open source permissioned distributed ledger technology (DLT) platform developed by R3 in 2016[17] [23]. Corda was originally designed to meet the needs of regulated financial institutions, but it has proved to be more widely applicable. It is heavily inspired by traditional blockchain systems, but Corda changed some design choices to make it more appropriate for real-world business transactions [16].

The main difference between Corda and a traditional blockchain is that in a traditional public blockchain network, anyone can join the blockchain network and can only be identified by a pseudonymous public address. But in Corda, each node represents a verified IP address that has passed through a stringent KYC process, and these nodes will be added to the network map. All nodes in this network map can transact with other nodes in a peer-to-peer manner through this map, and this communication is based on Transport Layer Security (TLS)-encrypted messages sent over AMQP/1.0 [31]. Fig. 3.1 from [1] shows a highlevel architecture of Corda. Next, I'll explain in detail how Corda works.

3.2.1 Corda Ledger

Unlike traditional blockchain systems, in Corda, transactions and ledger entries are not globally visible. Instead, each node maintains their individual database or ledger, Fig. 3.2 is a diagram of a node's internal structure from [12]. Transactions between different parties are only visible to them and only they will store the updated data in their databases. Because of this, if a user wants to do broadcast in Corda, he needs to send transactions to every node in the network. In addition, data updates are subject to contractual confirmation and validation by the Corda notary to confirm that it is a valid update. And we don't need to worry about a single point of failure of the notary because there are many notary nodes in the Corda network.

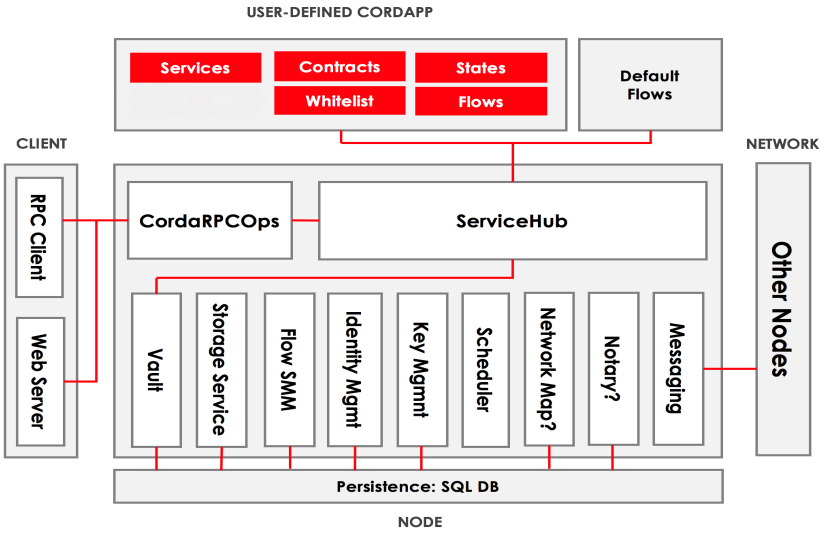


Figure 3.2: Node's internal structure [12]

3.2.2 State

"State object" is the fundamental building block in Corda which maintains the data storage. This is a digital document that records the existence, content and current status of an agreement between parties. The participant of the transaction will have access to the transaction information via the status and can update his database by update the corresponding state object stored.

3.2.3 Contract

The contract in Corda is similar to constraints on the update of the states, each state has its corresponding contract. Note that, despite their similar names, the contract in Corda differs from a smart contract. If your transaction contains a certain state, the transaction must comply with the contract corresponding to this state. A contract can specify many requirements of a transaction, such as whether this transaction requires or cannot have an input state, whether it should have an output state and whether it requires signatures from both parties. Figure 3.3 is an example of a state with corresponding contract from [16]. This state represents a deposit with an amount of 100 GBP, with its issuer and owner. And have references to its corresponding contract and legal prose.

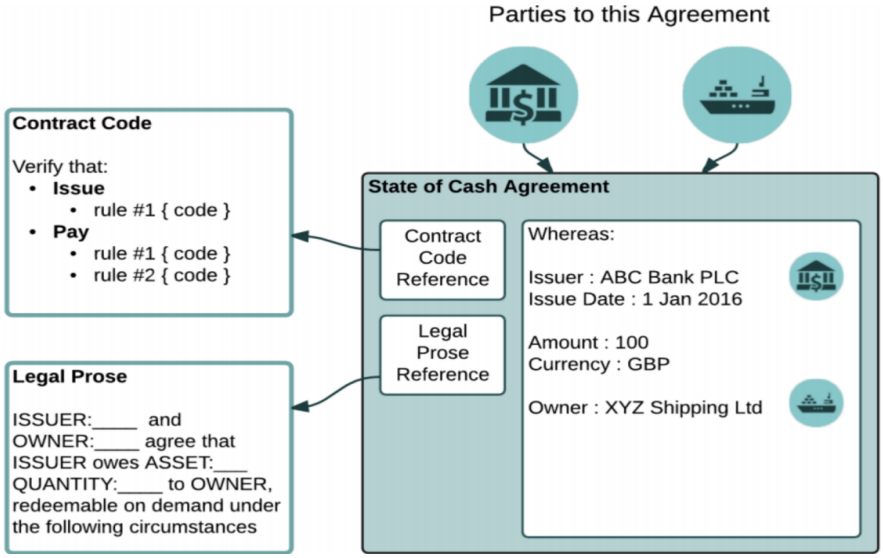


Figure 3.3: A state example [16]

3.2.4 Transaction

In Corda, transaction is the way to update ledgers. It will consume input states and produce output states. And the structure of the transaction is based on the UTXO model which can have a customised number of inputs and outputs. Figure 3.4 is an example of a transaction from [16]. This is a cash issuance transaction, which contains no input state and one output state. And contains the signature of the originator of the transaction which is the issuer bank.

3.2.5 Flow

Flow is a place to put specific steps used to implement a function. Flow, contract and state together composed Corda's smart contracts. And the flow is the most important part of it. It defines what operations need to be performed on a transaction, who the participants are, what state needs to be selected to store the data, and what input data it needs, in short, it is the driver of the functionality of the smart contract. Similar to

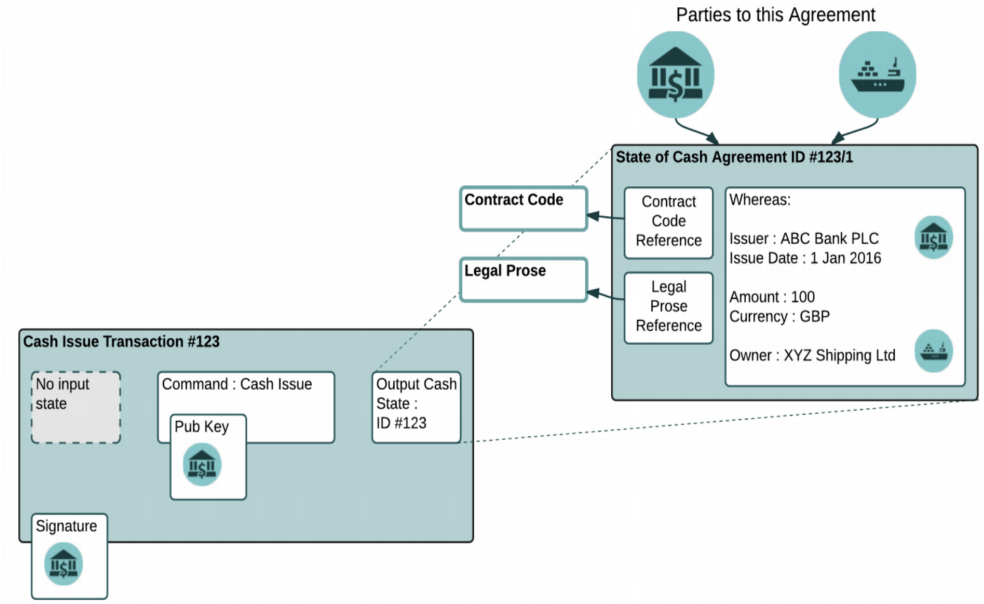


Figure 3.4: A transaction example [16]

calling a public function in Ethereum, users can initiate a flow to perform a specific function for them.

3.2.6 Consensus

Since Corda is a permissioned blockchain, only related parties and notaries will be involved in a transaction. It does not need to build a consensus from other nodes. But it does have 2 consensus for related participants.

- *Transaction validity*: All the participants can check associated contracts in a proposed transaction to ensure the output state of the transaction is valid, required signatures are included in this transaction, and other transactions involved in this transaction are also valid.
- *Transaction uniqueness*: In Corda, every input state in a transaction will be consumed after a transaction is done. So by checking whether the input state exists in the ledger, participants can be sure that all input states in this transaction exist only for this transaction and have not been consumed by previous transactions.

In addition to this, Corda has unique services that are ”pluggable” to increase scalability. Its individual services can also be coordinated by multiple untrusted nodes performing Byzantine fault-tolerance algorithms similar to those used in public blockchains. However, as this feature is not used in this project, it will not be described in detail here.

3.2.7 Notary

While both parties to a transaction can determine whether a transaction is valid by the two consensus points above, this does not protect against the double spending problem[19]. An attacker can attack by initiating two valid transactions at the same

time. So Corda introduces a notary mechanism to deal with this problem. Each transaction must involve a predetermined notary as an observer to determine which transaction comes first when several identical transactions happen in a short time. There are many notary nodes in the Corda network which can provide load balancing for higher transaction throughput. And each transaction can choose the nearest physically located notary node to reduce transaction latency.

Chapter 4

PCS Design

In this chapter, I first present an overview of my system design, including an analysis of the privacy leakage threat model, the security goals of the system, and a description of important implementation selections. Then, I provide a detailed design of how the system works through textual descriptions for the entire workflow and pseudo-code for several important actions. Finally, I describe potential attacks on the system and the corresponding defences.

4.1 Overall Design

4.1.1 System Framework

As shown in Figure 4.1, my system mainly involves four types of participants: *Requesters*, the *Primary Crowdsourcing platform*, the *Secondary crowdsourcing-organisation* (or sub-platform), and *workers/contributors*. In this section, I will describe who they represent and what they will do in the system.

- *Requesters*, identified by $R = \{R_1, R_2, ..., R_n\}$, is an individual or organization with task requirements (e.g., a passenger in ride hailing services).
- *Primary Crowdsourcing platform*, identified by P, refers to a third-party crowdsourcing service provider that offers a platform for requesters to post requests and workers or secondary organizations to accept tasks. The platform is responsible for managing the workflow of the crowdsourcing services, including task allocation, scheduling, data aggregation and analysis, quality evaluation of task execution, and reward allocation. It serves as the primary crowdsourcing organization.
- *Secondary crowdsourcing-organization*, identified by $P' = \{P'_1, P'_2, ..., P'_n\}$, is a secondary platform that can accept a task assigned by the primary organisation, and split this task again to sub-tasks and then assign the task to a set of selected workers managed by this organisation. Aggregate results from workers, return the final result to the primary platform and allocate rewards to contributors.

- *Workers*, identified by $W = \{W_1, W_2, ..., W_n\}$, is an individual participant in the system. Each worker may join one or multiple organisations, they act as contributors in this crowdsourcing system who accept and execute tasks, and gain reward from that.

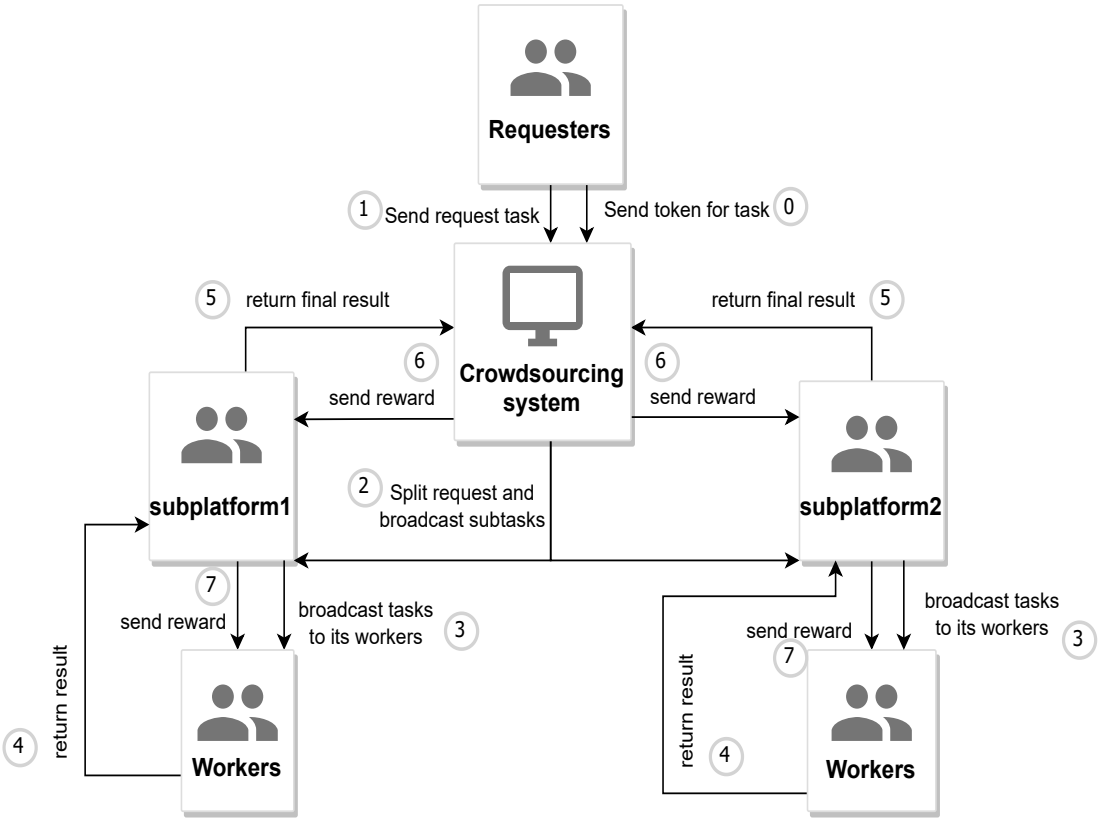


Figure 4.1: The system model of my crowdsourcing service system

4.1.2 Privacy Preserving Threat Model

This section talks about the threat model of my system that illustrates potential threats and malicious behaviour from each participants one by one. And indicates the security objective of this system. The project's goal is to achieve a crowdsourcing system with privacy protection, and here the main focus of our threat model is about privacy protection. More analysis of common attack methods will be covered in the Security Analysis section.

- *Requesters*: For requesters, here we assume they are all honest. The only thing requesters do in the entire crowdsourcing process, are to send the request and pay the tokens, they are not involved in the processing of the data, so we assume that requesters are honest.
- *Primary Crowdsourcing platform*: The primary platform is a potential attacker. Throughout the entire crowdsourcing process, the main platform is responsible for processing the most data. As it receives requests from requesters and distributes tasks to sub-platforms, it is inevitable that the platform will be aware of the

requesters' task requirements in order to enable the service to be performed. Additionally, since the main platform is responsible for calculating the rewards, it obtains the necessary data from the results of the sub-platforms. However, this process may leak unnecessary private data. For example, the sub-platforms' results may include details such as the task execution method, results of each task segment, degree of completion for each segment, and completion time. By analyzing large amounts of such data, the main platform could potentially obtain paticipants' other private information.

- *Secondary platform*: The secondary platform is another potential attacker. They are responsible for accepting tasks from the main platform and assigning them to workers. It is therefore possible for them to know the nature of the requester's request. Additionally, since they are responsible for aggregating and evaluating the results of the workers' work, as with the primary platform, they may be able to infer other private information from the workers' work results.
- *Workers*: Workers are also potential attackers. They are the performers of the tasks assigned by the sub-platform, so they have the potential to infer what the initial request is based on the task requirements and to analyse the demand preferences of the platform users through a large amount of data.

Let us then turn to our security objectives regarding this threat model. Our goal is to minimise the amount of data needed throughout the crowdsourcing process, while making it as impossible as possible to infer additional information from this data. Specifically, our feasible goal is to make it difficult for the main platform, as well as the subplatform, to know the specific details of task completion through the reward calculation process. And make the subplatform and worker difficult to know the goal of the main task that the sub-task they are doing serves.

4.1.3 Implementation Selection

Here is an overall description of the implementation choices of my schema, Workflow's details and more security analysis will be talked in the later part.

In order to mitigate potential privacy breaches during the crowdsourcing service, I have opted to deploy this service on a permissioned blockchain, which is Corda to minimise the number of observers per transaction. As outlined in the background chapter, data transfer in Corda is facilitated through peer-to-peer transactions, ensuring that only the transacting parties can view and store the data within the transaction. This feature effectively prevents other nodes within the Corda network from inferring private information via the intermediary data carried in the transaction.

Moreover, each transaction in Corda is required to adhere to the constraints specified in the transaction contract. Consequently, every successful transaction is guaranteed to be theoretically feasible. Simultaneously, I have employed Paillier homomorphic encryption for the data in certain transactions, which serves to further diminish the risk of privacy breaches.

Specifically, during the service process, both the main platform and the sub-platform

must receive the execution results returned by the workers for the tasks they publish and calculate the corresponding reward value. These execution results contain much information, including execution time, the working method utilized to complete the task, task type, and the completion ratio of the task content (since a task may encompass multiple requirements, the completion ratio refers to the collection of each requirement's completion ratio).To address the privacy concerns associated with these data, I employed Paillier homomorphic encryption to encrypt each individual data point, assigning a corresponding weight for the final result calculation. When the receiver evaluates the result, the result they get will be the product of this encrypted data. Because of Paillier's additive homomorphic property, after decrypting the product of all the ciphertexts, the result obtained is the total score of the task with each part's score distributed according to their respective weights. Through this approach, even the receiver remains unaware of the task's specific completion details, thereby reducing the potential for privacy leakage.

4.2 PCS Workflows

This section describes detailed workflows of my implementation in their execution order.

Step1: System initialization

1. Participants register.
2. Generate public-private key pair(pk_i, sk_i) corresponding to the blockchain address in accordance with the methods of the Paillier encryption mechanism(here I use an open source Paillier encryption library [29] for this).
3. made the public key pk_i of each participant public.

Step2: Request send, segmentation, and broadcast

1. Requester R_i send its request r and token for this request to primary platform.
2. Primary platform splits this request r to several tasks $r \rightarrow \{r_1, r_2, ..., r_n\}$.
3. Broadcast $\{r_1, r_2, ..., r_n\}$ to secondary platforms $\{P'_1, P'_2, ..., P'_n\}$.

Step3: Task accept, segmentation and broadcast

1. Secondary platform P'_i accept a task r_i from platform
2. Split the accepted task r_i to several sub-tasks $r_i \rightarrow \{t_{i,1}, t_{i,2}, ..., t_{i,n}\}$.
3. Broadcast sub-tasks $\{t_{i,1}, t_{i,2}, ..., t_{i,n}\}$ to its workers $\{W_1, W_2, ..., W_n\}$.

Step4: Task status update

1. After secondary platform P'_i accepted a task r_i, primary platform will change this task's status to "processing" so other secondary platforms can not accept it anymore.

Step5: Sub-task accept, status update

1. Worker W_i accept a sub-task $t_{i,1}$ from secondary platform P_i'.
2. Secondary platform P_i' update sub-task $t_{i,1}$'s status to "processing".

Step6: Execute sub-task, encrypt sub-task result and return result

1. Worker W_i execute the accepted sub-task $t_{i,1}$, get task completion result $d = \{d_1, d_2, ..., d_n\}$.
2. Before returning the result, for each component of the result of the sub-task, give each of them a weight $a_i \sim (0,1)$ and apply Paillier encryption on each of them $c_i = E_{paillier}((d_i * a_i), sk_{W_i})$.
3. Multiply the collection of sub-task result cipher text one by one to get the final result $c_{product} = \prod_{i=1}^{n} c_i$ and return it to P_i'.

Step7: Result evaluation

1. Secondary platform P_i' receives the cipher text of final result $c_{product}$.
2. Apply paillier decryption on the result to get plaintext of the sum of each component of the result after multiplying weights $d_{sum,i} = D_{paillier}(c_{product}) = \sum_{i=1}^{n} d_i * a_i$.
3. Secondary platform P_i' then wait for other sub-tasks' result.

Step8: Generate and return final task result

1. After receiving all sub-tasks' results, give each sub-task's result a weight $a_i \sim (0,1)$ and apply Paillier encryption on each of them $c_{task,i} = E_{paillier}((d_{sum,i} * a_{sum,i}), sk_{P_i'})$.
2. Return the product of the collection of task result cipher text $c_{product,task} = \prod_{i=1}^{n} c_{task,i}$ to primary platform P.

Step9: Primary platform's reward evaluation and sending

1. Primary platform P receives the cipher text of final result $c_{product,task}$.
2. Apply paillier decryption on the result to get plaintext of the sum of each result's completion ratio after multiplying weights $d_{task,i} = D_{paillier}(c_{product,task}) = \sum_{i=1}^{n} d_{sum,i} * a_{sum,i}$.
3. Calculate reward to secondary platform P_i' according to its task completion ratio and its calculation algorithm A, $reward = A(d_{task,i})$.
4. Send the reward to secondary platform P_i'.

Step10: Secondary platform's reward evaluation and sending

1. After the secondary platform receives the reward from primary platform P, it calculates each sub-task's reward according to its sub-task completion ratio and its calculation algorithm A', $reward_{sub-task,i} = A(d_{sum,i})$.
2. Send reward to each corresponding worker.

4.3 PCS Model Definition

In this section, in order to help readers better understand the workflow of my design. I describe in detail the operations performed by the 4 main actions in the workflows through pesudocode. These actions are: *Request sending*, *Receive request and broadcast tasks*, *Execute and return result* and *Result evaluation*.

Request sending. First, let's talk about request sending. This process is only used by requesters when they send their requests to the main platform. Algorithm 1 describes what we do during the sending of a request. We first initialize a transaction builder object and add the request to the builder as a transaction output state. Then, we use the contract corresponding to the request state to validate the transaction. If the validation passes, we send the request and pay the tokens.

Algorithm 1 Request sending

Input: Requester R_i, Task description T, Token id for paying token for this request $Token_id$, Platform P
Output: Boolean of request successfully sent or not.
Initialize $Request \leftarrow newRequest(R_i, T, Token_id, P)$
Initialize $Transaction \leftarrow newTransaction(R_i, P)$
Initialize $Token \leftarrow newToken(Token_{id}, R_i)$
$AddOutputState(Transaction, Request, RequestContract)$
if $Verify(Transaction, RequestContract)$ pass **then**
  if $checkOwner(S, Token)$ pass And $checkAmount(Token, A)$ pass **then**
    $Send(Token, P)$
    $Send(Transaction)$
    $Store(Transaction.Request)$
    return true
  else
    return false
  end if
end if

Request reception and tasks broadcasting. Both primary platforms and sub-platforms need to receive tasks and broadcast sub-tasks. Although the details of their implementations are different, their general ideas are the same. As shown in Algorithm 2, they first receive the request from another party and verify it according to the corresponding contract. If verification passes, they split the task into sub-tasks using their allocation algorithm. Then, they create a transaction builder, put all sub-tasks as output state, verify it, and broadcast the transaction to all sub-platforms or workers.

Task execution and result returning. Both sub-platforms and workers may use this flow, but there is a difference in how they handle results. For workers, the results they send are from their own execution, while sub-platforms collect sub-results returned from workers. Algorithm 3 is applicable to the scenario faced by workers, workers first receive a task content transaction. If the verification of the transaction passes, they store the task requirements and begin execution. After the execution finishes, they create a

Algorithm 2 Request reception and tasks broadcasting

Input: Requester R_i, transaction $Transaction$,Platform P,sub-platforms $P'_1, ..., P'_n$
Output: Boolean of successfully broadcasted or not.
$Receive(Transaction)$
if $Verify(Transaction, RequestContract)$ pass **then**
  $Store(Transaction.Request)$
  $Task_{list}(T_1, ..., T_n) \leftarrow Split(Request)$
  Initialize $Transaction' \leftarrow newTransaction(P, (P'_1, ..., P'_n))$
  $AddOutputState(Transaction', Task_{list}(T_1, ..., T_n), TaskContract)$
  if $Verify(Transaction', TaskContract)$ pass **then**
    $Broadcast(Transaction', P'_1, ..., P'_n)$
    return true
  else
    return false
  end if
end if

list of result elements, apply Paillier encryption to each element, calculate the product of the cipher texts as the final result, and then create a new transaction to return the final cipher text of the result.

Algorithm 3 Task execution and result returning

Input: Worker W, transaction $Transaction$,secondary platform P'
Output: Boolean of result successfully returned or not.
$Receive(Transaction)$
if $Verify(Transaction, sub-taskContract)$ pass **then**
  $Store(Transaction.subtask)$
  $result_{list}(R_1, ..., R_n) = Execute(subtask)$
  $Results'ciphertext(c_1, ..., c_n) = Encrypt_{paillier} result_{list}(R_1, ..., R_n)$
  Initialize $Transaction' \leftarrow newTransaction(W, P')$
  $product \leftarrow Product(Results'ciphertext(c_1, ..., c_n)$
  $AddOutputState(Transaction', product, subtaskContract)$
  if $Verify(Transaction', subtaskContract)$ pass **then**
    $Send(Transaction')$
    return true
  else
    return false
  end if
end if

Result evaluation. Both the primary platform and sub-platform need to perform result evaluation, and the process they follow is similar. As shown in Algorithm 4, they receive task result transactions from workers/sub-platforms, store the encrypted results, and use Paillier decryption on the cipher text to obtain the final score. They then send tokens to workers/sub-platforms based on the final score.

Algorithm 4 Evaluate result

Input: sub-platform P', transaction $Transaction$,Platform P,Token id $Token_{id}$
Output: Boolean of successfully sent reward or not.
$Receive(Transaction)$
if $Verify(Transaction, ResultContract)$ pass **then**
  $product \leftarrow Transaction.product$
  $finalResult \leftarrow Decrypt_{paillier}(product)$
  $reward \leftarrow Evaluate(finalResult)$
  if $checkOwner(P, Token)$ pass And $checkAmount(Token, reward)$ pass **then**
    $Send(Token, P')$
    return true
  else
    return false
  end if
end if

4.4 Security analysis

In this section, I will talk about various potential hazards and vulnerabilities that may arise in this type of system, as well as the security measures implemented to mitigate these risks.

4.4.1 DDoS attack

Distributed denial of services(DDoS) attack [37] is a type of cyberattack that uses distributed multiple systems to flood a target system. Using overwhelming amounts of traffic to disrupt the victim system's normal functioning (either by overloading it or by permanently hogging a resource), prevents legitimate users from using the system properly.

In our system, there are two primary methods for implementing Denial of Service attacks. The first is by sending a large number of task requests to the main platform, occupying the majority of processing resources and preventing other users' requests from being executed promptly. The second method involves workers or sub-platforms accepting tasks but not executing them, causing the tasks to be indefinitely stalled and unable to progress normally.

Since our design lets users pay tokens for their requests in advance, if an attacker chooses to carry out a Denial of Service attack in the first way, the cost incurred would likely be prohibitively high, and maintaining the attack for an extended period would be generally unfeasible. As a result, the likelihood of our system being targeted by such an attack is extremely low.

Regarding the second method, in my design, all workers and sub-platforms are registered individuals or organizations with a high level of trustworthiness. Moreover, even if they do accept tasks but do not execute them, the main platform can resolve this issue by reassigning the task and revoking the worker's eligibility to work on tasks. This

approach ensures that tasks continue to progress and that any uncooperative executors are removed from the system.

4.4.2 Double spending attack

Double spending attack is a well-known problem for digital currencies [20]. In this attack, the attacker creates two transactions within a short time using the same digital tokens. The objective is to have the first transaction's tokens used as payment for the second transaction before they are deducted from the initial transaction. If successful, this would result in both transactions being confirmed, effectively allowing the attacker to spend the same tokens twice.

In my design, the service is developed based on the Corda platform, which employs a consensus mechanism centred around notary nodes [8]. Notary nodes are multiple clusters consisting of one or multiple notaries, and each transaction requires a signature from a notary node. The notary nodes are responsible for checking the uniqueness of each transaction. If a notary node discovers that an input state in a subsequent transaction has been used in another transaction, it will reject the new transaction, effectively preventing double-spending attacks from occurring.

4.4.3 False reporting and free riding attack

False reporting and free riding attack [45] in my system manifests as workers submitting fake results, pretending they have completed the work, and receiving payment from the requester without actually performing the task.

To address this issue, I have adopted a solution that involves carefully defining each transaction contract within Corda to minimize the likelihood of such attacks succeeding. By defining restrictive conditions and validation steps in the contract, we can ensure that only legitimate work results are approved by the contract. Additionally, due to the immutability of the blockchain, even if a misjudgment occurs, it is easy to trace the specific worker node responsible for the deceptive behavior from the recorded information and impose appropriate penalties on the offending party.

4.4.4 Data loss

In traditional crowdsourcing systems, data loss is a potential issue that could lead to significant consequences. Since data is normally stored by a trusted third-party (normally by the main platform), and any accidents (such as an employee's mistake) or force majeure events (like fires or earthquakes) that cause data loss can severely impact the normal operation of the service. However, in my design, all data is stored on a distributed ledger, which effectively prevents the aforementioned problems from occurring in my system.

Chapter 5

Implementation and Testing

In this chapter, I will provide a detailed description of the code implementation of my crowdsourcing system, which is also this project's main contribution, including the overall code architecture and the specific realisation of important flows.

Before delving into the specifics of my code, it is essential to understand how the Corda application operates and how developers develop applications based on the Corda framework. The Corda application's functionality relies on four main components: *state*, *transaction*, *contract*, and *workflows* [2]. Since their respective definitions and functions have been outlined in the background section, here I just give an overview of how they work together to make the application run and provide some code related examples.

A Corda application can be developed by Java or Kotlin, and this project uses Java [3]. In Corda, state is the object that is used for data storing and data transfer. Specifically, as shown in figure 5.1, state is a self-defined Java class that implements one state type in the Corda library, and data is stored in its variables. Each state needs to be linked to a contract it belongs to.

```
@BelongsToContract(requestContract.class)
public class request implements LinearState {
```

Figure 5.1: First 2 lines of a state object

A contract is a self-defined Java class that implements the Corda library's abstract Contract class. It is mainly used to verify whether a transaction satisfies the restrictions defined in this contract, figure 5.2 are some example restrictions of a token issue contract.

```
private void verifyIssue(LedgerTransaction tx){
    if(tx.getInputs().size() != 0)
        throw new IllegalArgumentException("Zero Inputs Expected");

    if(tx.getOutputs().size() != 1)
        throw new IllegalArgumentException("One Output Expected");

    if(!(tx.getOutput( index: 0) instanceof tokenState))
        throw new IllegalArgumentException("Output of type tokenState Expected");

    tokenState tokenState = (tokenState)tx.getOutput( index: 0);
    if(tokenState.getAmount() < 1)
        throw new IllegalArgumentException("Positive amount expected");
```

Figure 5.2: Constraint examples

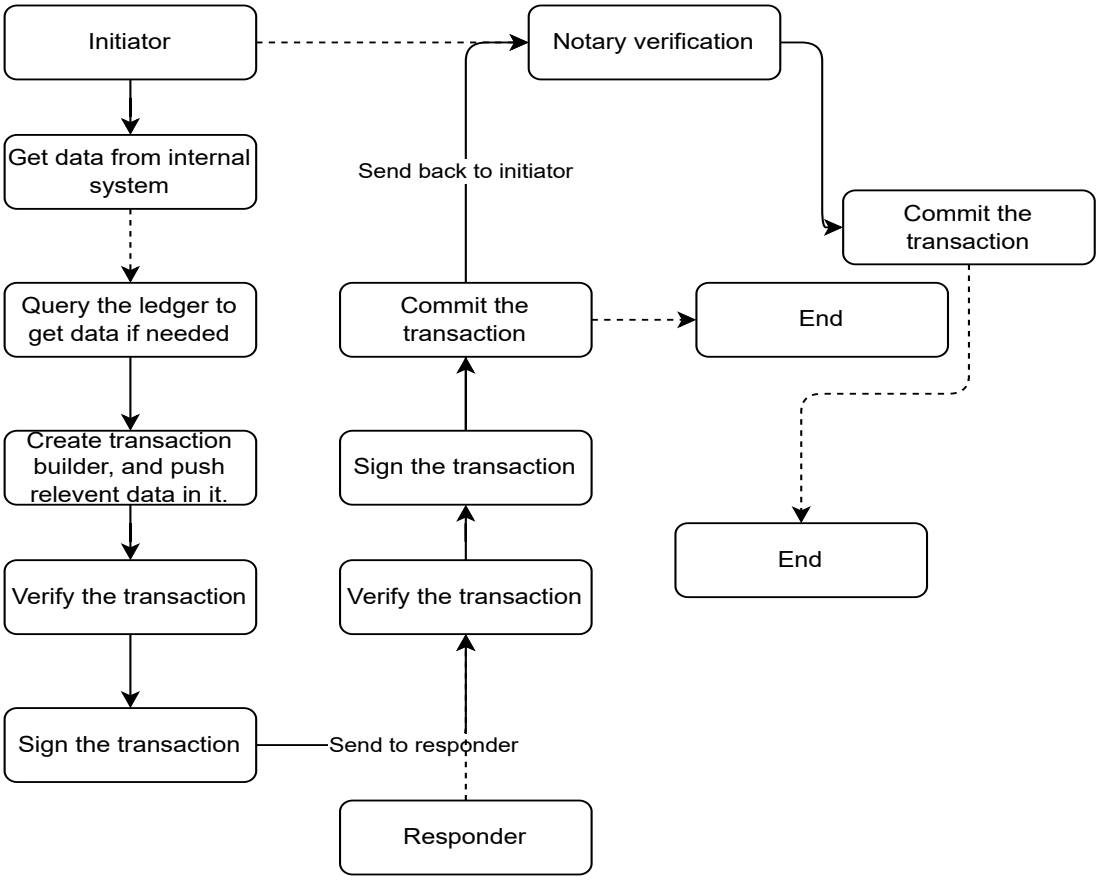


Figure 5.3: Flow chart of a typical flow

A transaction is an object used to update the Corda ledger, with the input and output states stored in the transaction object. It is typically initialized and called within a workflow. The implementation of the Corda application's functions is mainly developed in workflows. A flow defines the actions that a node in the Corda network can call. A flow class usually consists of two main parts: the initiating flow and the flow responder.

As shown in Figure 5.3, the initiating flow begins by initiating the transaction builder and incorporating relevant input/output states. It then verifies the transaction in accordance with the contracts, signs it, and forwards it to the responder. The responder then proceeds to verify, sign, and commit the transaction, sending the fully signed transaction back to the initiator. Upon receiving the signed transaction, the notary will verify the fully signed transaction. If the verification is successful, the initiator also commits the transaction, and both parties update their ledger with the data changes resulting from the transaction [6].

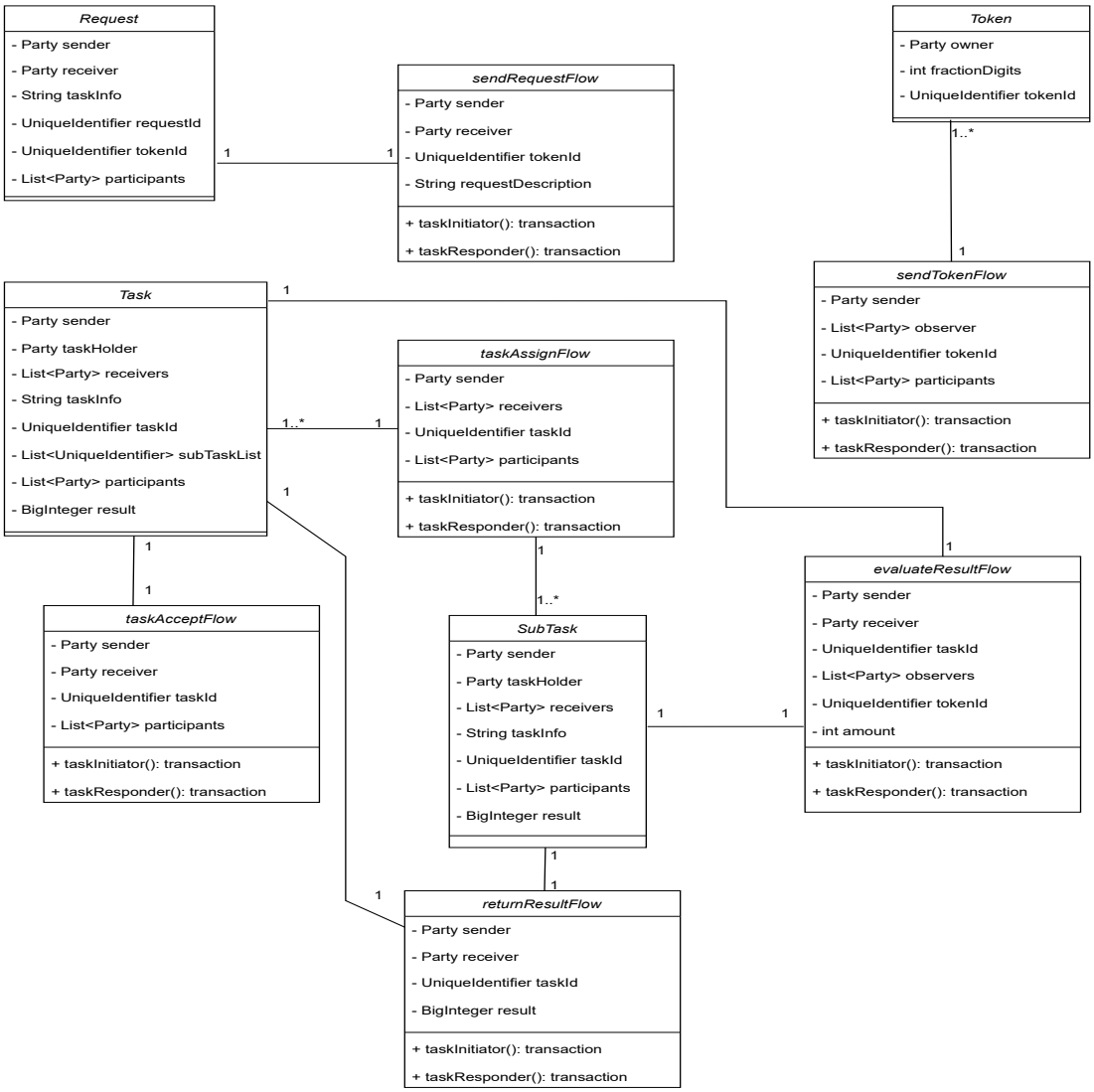


Figure 5.4: UML class diagram for this system

5.1 Program Overview/framework Diagram

Fig. 5.4 is an UML class diagram for my system, in order to make it looks more clear, the diagram has been simplified to a certain extent, but it still offers a general overview of the system's structure.

5.2 Implementation Details

In this section I will provide a detailed overview of the state, contracts, and flows of my code. By reading the responsibilities of all main components, the reader can understand how the whole system works.

5.2.1 States and Their Contracts

In my system, I have 4 main token types: request, Task, subTask, token.

- *Request state* is used in the request sending stage. As shown in Figure 5.5, it stores the essential information required for request processing. Every request has a unique identifier, ”requestId”, which is used for state query in the Corda ledger. By using a unique ID in the implementation of a flow, a node can query its own database according to this ID and fetch the state object from the ledger for further processing. Each request’s requirement content is stored in ”taskInfo”. The platform can use this information to determine how to split the request into tasks later. The list of parties, ”participants”, stores every party involved in this transaction. This list is used in the signature collection part that is required in a flow’s transaction verification. The unique ID, ”tokenId”, is used to identify which token the requester sends to pay for the request.

  The contract defined for this request state is called ”requestContract”. It contains essential restrictions for the request transaction, for example, a request transaction should contain 0 input state and 1 output state, which is the request state. All required fields should not be empty, etc.

- *Task state* is used to represent a sub-task of a request. Each task has a taskHolder party, which is the sub-platform that accepts this task. This field is initially null in the task broadcast phase and changes to the information of a sub-platform after it has accepted the task. The list ”subTaskList” records the unique ID of each work task involved in this sub-task. It will be called in the result aggregation phase and reward sending phase later. The BigInteger ”result” is where the ciphertext of the final result is stored. The other variables are similar to those in the request state, so they will not be further elaborated here. The contract defined for the Task state is called ”taskContract”. Its purpose is also similar to the previous contract.

- *SubTask state* is a modified task state that changes the adaptation scenario from the main platform and the subplatform to the subplatform and its workers, with no major changes to its role or implementation.

- *Token state* is used to represent fees payment in our system. This state extends the Corda library’s EvolvableTokenType state, and most of its functions are already implemented by the library. Thus, my implementation only needs to set a few essential variables and functions. As shown in Figure 5.6, it contains three private variables: owner, fractionDigits, and tokenId. The owner and tokenId are used to identify the owner of the token and the token itself, while fractionDigits refer to the minimum number of digits into which the token can be divided. The toPointer function, with the help of the library, returns the pointer of this token, which will

be used in later token transfer and division. Details about Corda's token SDK can be found at [10].

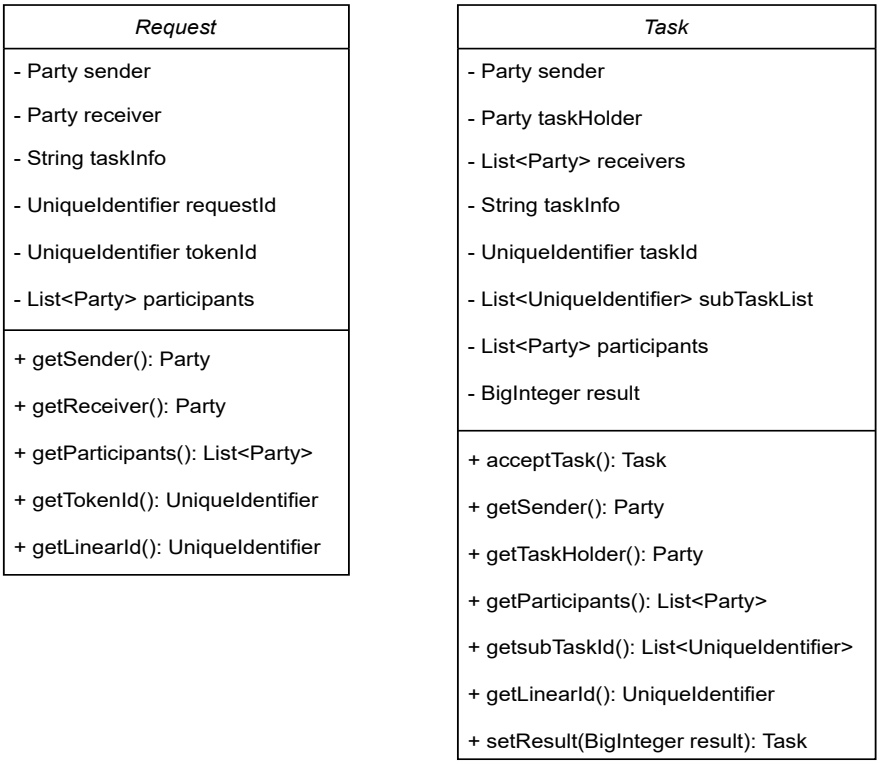


Figure 5.5: Request state and subTask state.

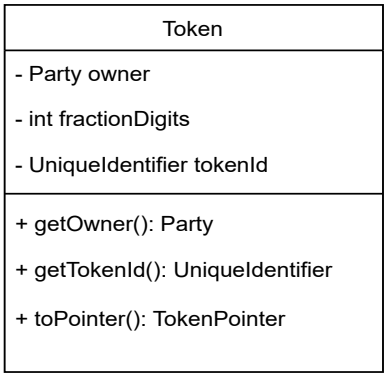


Figure 5.6: Token state

5.2.2 Workflows

The proper functioning of my system uses the following workflows: issueTokenFlow, sendTokenFlow, requestFlow, splitTaskFlow, acceptTaskFlow, assignWorkFlow, acceptWorkFlow, returnWorkResultFlow, returnResultFlow, evaluateFlow and sendWageFlow. I will explain them one by one.

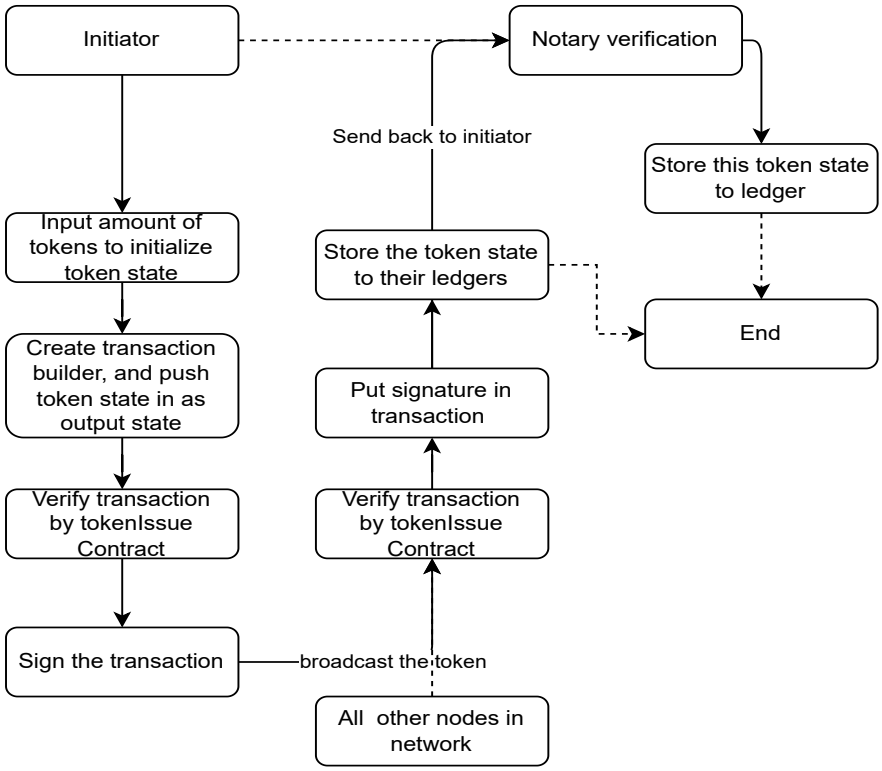


Figure 5.7: Flowchart for issueTokenFlow.

IssueTokenFlow in my system can be initiated by any node that enters the desired number of tokens to be generated for testing purposes. As shown in Figure 5.7, once initiated, the generated token state is sent to all other nodes within the Corda network. This allows the network to recognize the existence of the newly-created tokens, making them available for trading. Additionally, each node stores the token data on their respective ledgers for further processing and tracking. Each involved party in a transaction consumes the input state and changes it to the output state to achieve the ledger update. The contracts used for the transaction and the signatures of the participants are also stored in this transaction object. In the Corda network, communication between various nodes is achieved through the processing of transaction objects.

SendTokenFlow can be initiated by any node to transfer their token to another party. The workflow process is similar to the issueTokenFlow, as it also needs to create a transaction and broadcast it to the network to let other nodes acknowledge this token transfer. However, the main difference is that we have an input state which is the original token state used for token transfer. As shown in Figure 5.8, if Bob wants to send 2 tokens to the platform, he first needs to push his token state used for the token transfer to this transaction. He then needs to input 2 output states: the first is the original token after subtracting the amount of 2, and the other is the token state owned by the platform after the token transfer. So after this transaction, the original input state is consumed and generates 2 new output states corresponding to this transaction result.

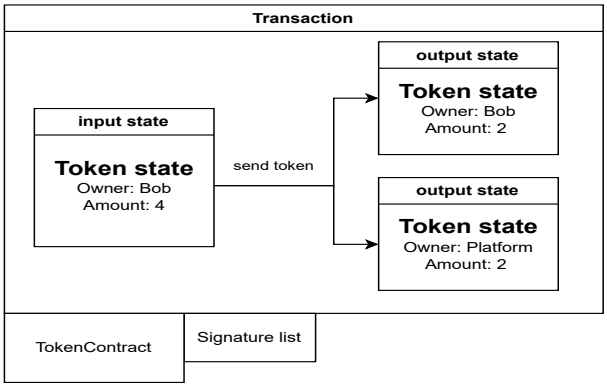


Figure 5.8: Transaction for token transfer.

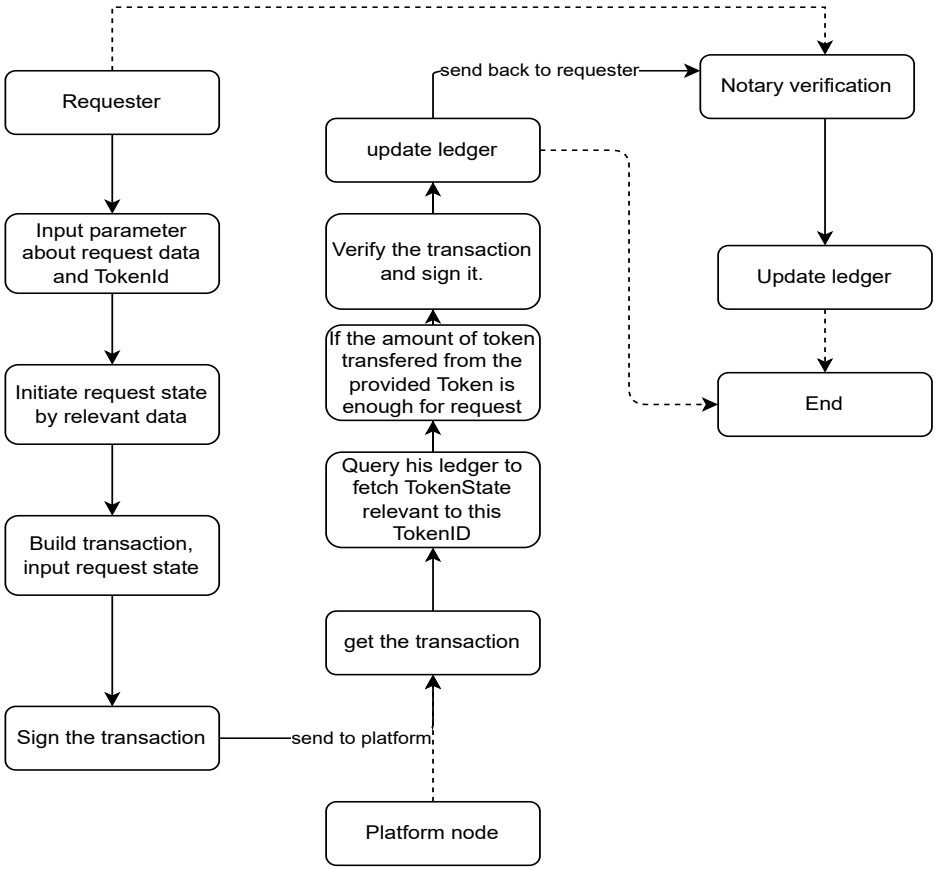


Figure 5.9: Flowchart for request sending.

RequestFlow is initiated by a requester to send a request state to the platform, in this flow, the requester creates a request state object with relevant information, but before sending this request, he needs to do a token transfer to pay for the request before activating requestFlow. The transaction structure for requests is similar to previous transactions but the workflow's algorithm has some changes. As shown in figure 5.9, the initiator needs to provide a tokenID along with the request state, at the responder side, besides using the contract to verify the transaction, he also needs to verify if the

requester has paid enough tokens for the request. What he does is query from his ledger, fetch tokens that get transferred from the given tokenID, check the amount and sender identity, if everything is valid, then verify the transaction and push to the next stage.

SplitTaskFlow is initiated by the platform to split the given request into multiple tasks and broadcast these tasks to his sub-platforms, and these task states will be stored in every participant's ledgers for progress tracking and further processing. But here, the platform does not use the request state as the input state, there are 2 reasons, the first is he still needs to do subsequent processing of the request state, and if this state is consumed at this step, the information about this state is invalidated. The other reason is keeping the information available to each participant as small as possible for privacy protection was a fundamental requirement of our development, sub-platform does not need to know the request information for task processing, so they are not allowed to see it.

AcceptTaskFlow is initiated by sub-platforms to accept the broadcasted tasks. In this transaction, the input state is the task state initiator wants to accept, he uses the TaskId, query his ledger to fetch the task object, and call its public function "acceptTask()" to try to change the taskHolder to himself. If the taskHolder is currently null, he can successfully change it and use the changed task object as the output state, broadcast the new state to other participants to let the network acknowledge he is the taskHolder for task executing now.

AssignWorkFlow is similar to splitTaskFlow, when a sub-platform accepts a task, he will call this flow to allocate multiple work tasks to his workers. And for each sub-work, he will generate a workState for further processing and progress tracking.

AcceptWorkFlow is similar to acceptTaskFlow, this flow just allows a worker to accept a sub-task to become the executor of it.

ReturnWorkResultFlow is initiated by the worker who is executing a work task. In our framework, as shown in figure 5.10, after executing a work, the worker will have multiple relevant data about the work result, for example work execution method, results of each work segment, degree of completion for each segment ,etc. Before returning a final result, workers will first recalculate the results by weighting each sub-data, then apply Paillier encryption on each result segment, and calculate the product of these cipher text. And the final product of cipher text is the data we will put in the result field. At the responder side, since the initiating flow already applied weights for result calculating, what they do is simple apply a Paillier decryption on it and get the final result score, then both parties will update this change to the ledgers and the status of this work will be marked as completed.

ReturnResultFlow is just another version of returnWorkResultFlow. At this flow, the sub-platform encrypt each sub-work of a task, and calculate their product to get the final result of this task.

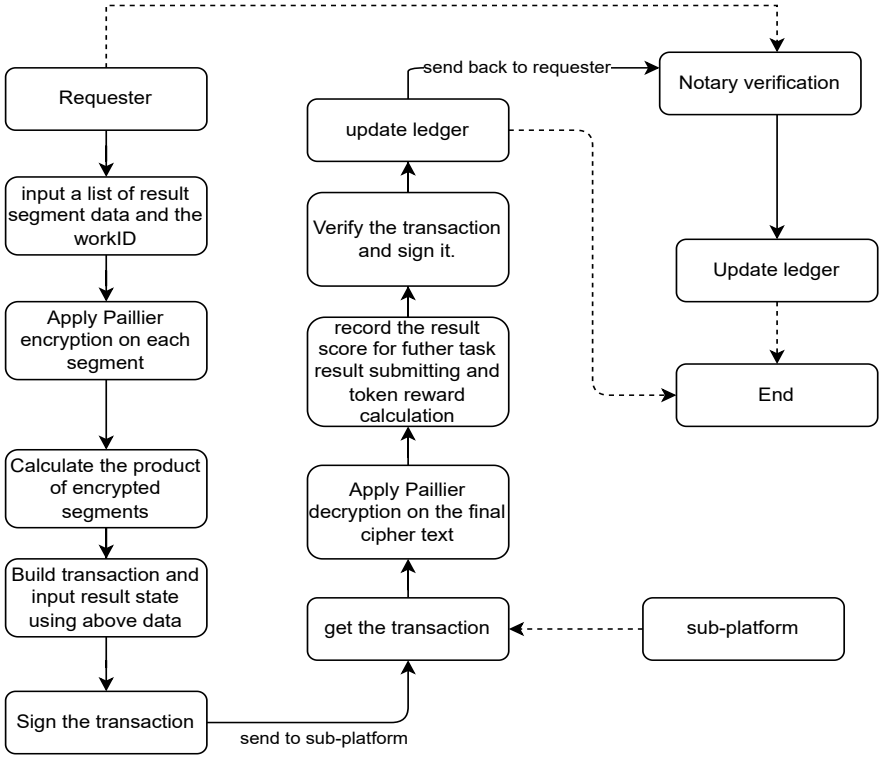


Figure 5.10: Flowchart for result returning.

EvaluateFlow is initiated by main platform to evaluate a task and send reward according to the completion result. He will query his ledger to fetch the task object that he wants to evaluate according to taskID, and the result field of the task object will be the cipher text of final result returned by sub-platform, if the result is null, it means the task has not been done, then it will return false. Otherwise, platform will use paillier decryption to get the final score of this task. Then the platform will send tokens to the sub-platform according to his task's completion score, and all participants will update relevant information about this transaction.

SendWageFlow is initiated by sub-platform after main platform sent task reward to him. This flow needs the input of Token used for wage sending and TaskId that represents which task's executors he is paying for. The initiating flow will query his ledger by TaskId to fetch the task state object which contains a list of sub-tasks's ID, then he will query each sub-task one by one, get their completion score, and pay them proportionally according to their score.

5.3 execution test

Fig. 5.11 shows this application running in the Corda test environment. As you can see, each node will have its own shell command window, and most of my tests are performed by having each node call flows through these windows.

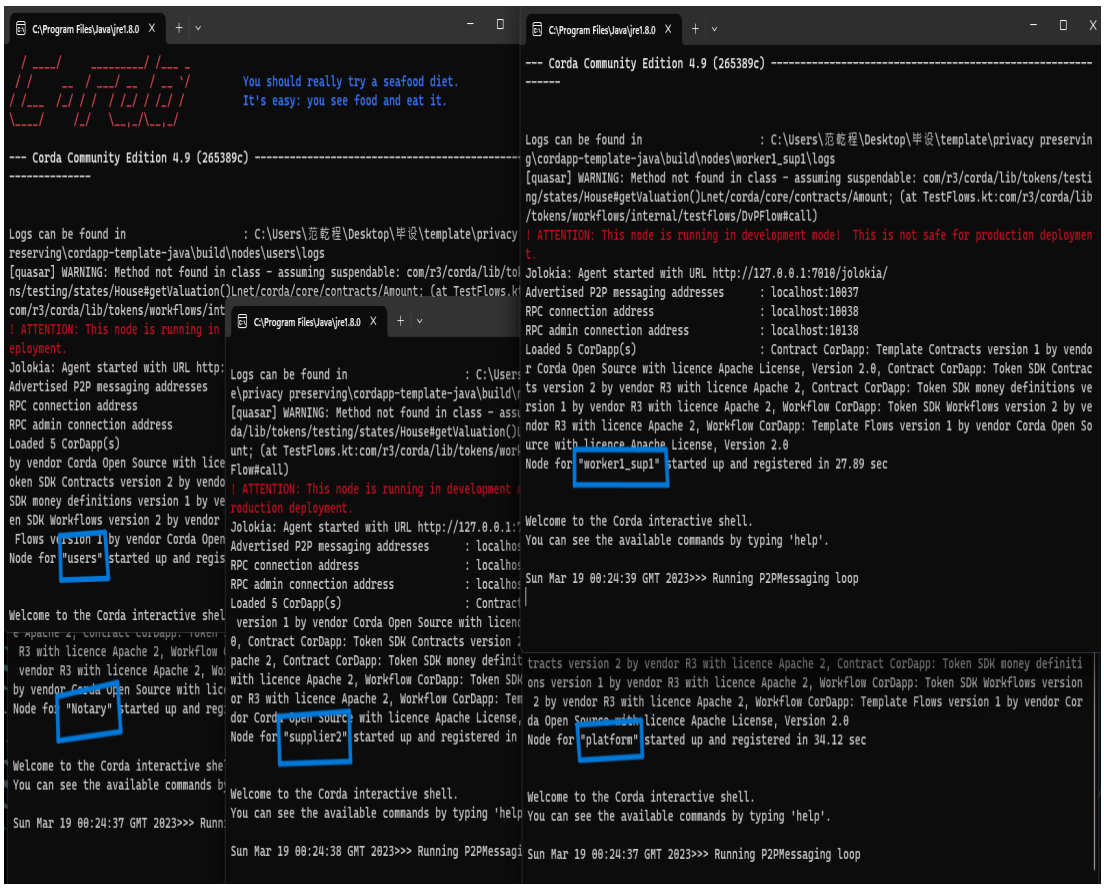


Figure 5.11: Execution test

Fig. 5.12 is an example of a transaction call to show how the flow is called and what messages the user will see, more call scenarios for other flows will be shown in appendix A, and more details about the flow performance will be shown in Chapter 6.

```
start createToken msg: "valid", amount: 4
Starting
Done
Starting
Done
Starting
Done
Flow completed with result:
Message: valid
Token Id is: a7af901b-9b9d-4671-9fc8-d97710463986
Storage Node is: [O=users, L=New York, C=US]
amout is: 4 TokenPointer(class com.template.states.CustomTokenState, a7af901b-9b9d-4671-9fc8-d97710463986) issued by users
 time taken is: 0.8808415
```

Figure 5.12: createTokenFlow

Chapter 6

Performance Evaluation

This section presents experimental data on the system's performance and analyzes the impact of modifications to variables such as participating nodes and transmission message size on the system's overall performance.

6.1 Design of experiments

In this section, the impact of various variables on the results of the system is examined from three distinct perspectives by means of three sets of experiments.

The first set of experiments tested the effect of changing the number of sub-platforms and working nodes on the system performance while holding all other variables constant.

The second set of experiments focuses on the size of messages transmitted by each transaction. Specifically, for each test, a string of task/request descriptions with varying lengths is selected and tested for its impact on system performance. In this experiment, the number of sub-platforms and workers was held constant, with two sub-platforms, each with one worker.

The last set of experiments focuses on the influence of Paillier encryption on the system. I tested the system separately with and without encryption steps with Paillier encryption to check the impact of this mechanism on execution time.

The present experiment considers the execution time as the duration required for completing the entire crowdsourcing process. Specifically, this process comprises several workflows, beginning with a user requesting a task on the platform. Following this, the platform divides the task into two sub-tasks, which are then assigned to sub-platforms. The sub-platforms, in turn, further divide the sub-tasks into two additional sub-tasks each, and assign them to workers. Once the workers have completed the sub-tasks, they send the results back to the sub-platforms, which then compile the overall task result and return it to the platform. Finally, the platform evaluates the two task results and disburses payment to the sub-platforms. The cumulative time required to complete all of the above workflows is considered the final execution time in the context of our experiments.

6.2 Experiment Configuration and Performance Metric

All the following experiments were carried out in corda's local test environment. Here we use Corda with version 4.9, and other component's version numbers are all shown in Fig. 6.1. This test environment looks as shown in Fig. 5.11, as I mentioned before, each node has a shell command window and I can call flows through these windows to run workflows and implement experiments. Each node's information can be edited in the project's build.gradle file. Fig. 6.2 is an example of the main platform node's setting. Each node's name and port address can be edited in this way.

For the performance indicators of this experiment, there are 2 main parameters:

- T: the total time used to perform a complete crowdsourcing process(here, we call it completing a crowdsourcing transaction). We get this value by calculating the sum of time taken for all workflows that are involved in this crowdsourcing process.
- TPS(1/T): the number of crowdsourcing transactions can be performed in 1 second. We get this value by dividing 1 by T.

For T, the lower it is, the better the performance is. For TPS, the higher it is, the better the performance is.

```
cordaReleaseGroup=net.corda
cordaCoreReleaseGroup=net.corda
cordaVersion=4.9
cordaCoreVersion=4.9
gradlePluginsVersion=5.0.12
kotlinVersion=1.2.71
junitVersion=4.12
quasarVersion=0.7.13_r3
log4jVersion=2.17.1
platformVersion=10
slf4jVersion=1.7.30
nettyVersion=4.1.68.Final
```

Figure 6.1: Version information

```
node {
    name "O=platform,L=London,C=GB"
    p2pPort 10005
    rpcSettings {
        address("localhost:10006")
        adminAddress("localhost:10046")
    }
    rpcUsers = [[ user: "user1", "password": "test", "permissions": ["ALL"]]]
}
```

Figure 6.2: Node setting example

6.3 Measurement and Analysis

The first set of experiments aimed to investigate the impact of the number of participants on the system's performance. Due to limitations of Corda's local test environment, we

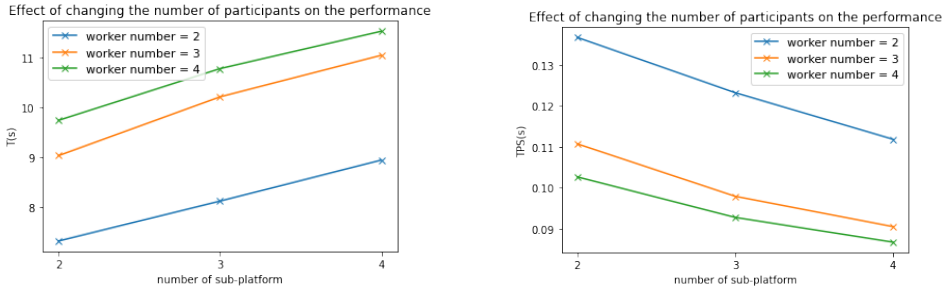
were unable to test the system with a large number of nodes as the environment would easily crash when the number of nodes is too large. Nonetheless, even with a limited number of nodes, we were able to discern a general trend in the results. Table 6.1 shows the system performance test results for varying numbers of participants, where P represents the number of sub-platforms, W represents the number of workers, and there is always one main platform, the request's description message always contains 5 characters. For example, 2P2W means there are two sub-platforms and two workers.

FLOW STEP	2P2W	3P2W	4P2W
REQUESTFLOW	0.309	0.315	0.265
SPLITTASKFLOW	0.109	0.110	0.093
ACCEPTTASKFLOW	0.701	1.024	1.273
ASSIGNWORKFLOW	0.107	0.134	0.146
ACCEPTWORKFLOW	0.122	0.123	0.124
RETURNWORKRESULTFLOW	0.387	0.365	0.387
RETURNRESULTFLOW	1.178	1.201	1.296
EVALUATEFLOW	0.891	1.136	1.116
SUM	7.316	8.118	8.944
FLOW STEP	2P3W	3P3W	4P3W
REQUESTFLOW	0.311	0.314	0.319
SPLITTASKFLOW	0.112	0.114	0.102
ACCEPTTASKFLOW	0.690	1.047	1.264
ASSIGNWORKFLOW	0.166	0.173	0.171
ACCEPTWORKFLOW	0.224	0.231	0.225
RETURNWORKRESULTFLOW	0.554	0.572	0.581
RETURNRESULTFLOW	1.221	1.324	1.511
EVALUATEFLOW	9.813	1.142	1.174
SUM	9.034	10.216	11.053
FLOW STEP	2P4W	3P4W	4P4W
REQUESTFLOW	0.296	0.304	0.309
SPLITTASKFLOW	0.117	0.104	0.108
ACCEPTTASKFLOW	0.780	1.088	1.288
ASSIGNWORKFLOW	0.186	0.185	0.186
ACCEPTWORKFLOW	0.271	0.254	0.257
RETURNWORKRESULTFLOW	0.650	0.708	0.675
RETURNRESULTFLOW	1.232	1.399	1.630
EVALUATEFLOW	1.192	1.185	1.186
SUM	9.744	10.784	11.537

Table 6.1: Execution time(s) for variable number of sub-platforms and workers.

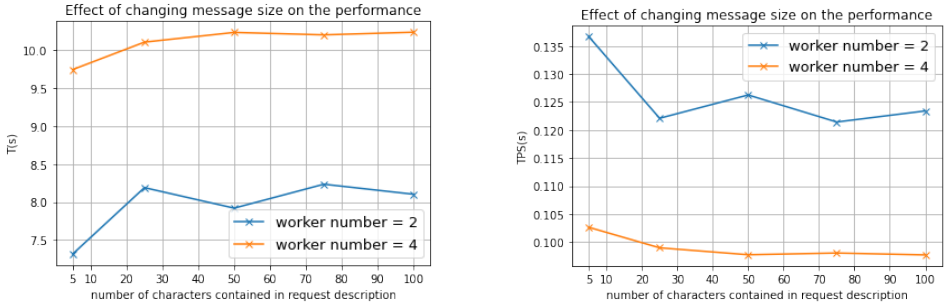
From the table, we can see that the performance of requestFlow remains consistent. This is because requestFlow only involves two nodes, the requester and main platform, and the number of workers/sub-platforms should not affect its performance.

For the other flows, we can categorise them into two groups: those that can theoretically be impacted by the number of sub-platforms(splitTaskFlow, acceptTaskFlow,returnResultFlow,evaluateFlow) and those that can theoretically be impacted by the number of workers(assignWorkFlow,acceptWorkerFlow,returnWorkResultFlow).



(a) Performance graph of T(s) (b) Performance graph of TPS

Figure 6.3: Performance graph of participants number experiment



(a) Performance graph of T(s) (b) Performance graph of TPS

Figure 6.4: Performance graph of message size experiment

In the second case, we observe a clear correlation between the number of workers and the execution time of each flow: as the number of workers increases, the execution time also increases. In the first case, however, one flow did not perform as expected, which is splitTaskFlow. Other flows' execution time all increase with the increase of the number of sub-platforms, but this flow always stays at the same level. My guess for this result is this, what makes the splitTaskFlow different from the others is that it has no input state, what it does is broadcast the split task out and does not require any input state, so the sub-platforms that are the receivers do not need to query their respective ledgers to get the data of the input state, what they do is simply record the output state to their ledgers. It makes sense that the amount of computation required to write data to the end of the ledger is much less than traversing the ledger looking for state, so the total increase in computation from increasing the number of sub-platforms by a small amount cannot have a large enough impact on the total running time in this experiment.

But in general, we can see from Fig. 6.3 that increasing the number of workers or subplatforms both lead to an increase in total running time and a decrease in TPS.

The second set of experiments aimed to investigate the impact of input message size on the system's performance. Table 6.2 presents the results of these experiments. Since input messages are only used in request submission and task splitting, we recorded only the total execution time and the execution time of the involved flows.

Overall, the results show that increasing the message size does lead to a slight increase in the execution time of requestFlow and splitTaskFlow. However, the difference is not

significant. For example, increasing the input message size by 2000% only increased the runtime of requestFlow by about 85% and increased the runtime of splitTaskFlow by about 220%. Moreover, since both processes take very little time to run, Fig. 6.4 indicates that changing the input message size does not have a big impact on the total time.

2P2W	5C	25C	50C	75C	100C
SUM	7.316	8.190	7.92	8.235	8.103
REQUESTFLOW	0.309	0.397	0.425	0.512	0.570
SPLITTASKFLOW	0.109	0.143	0.173	0.211	0.242
2P4W	5C	25C	50C	75C	100C
SUM	9.744	10.105	10.233	10.201	10.236
REQUESTFLOW	0.296	0.335	0.325	0.415	0.548
SPLITTASKFLOW	0.117	0.241	0.201	0.176	0.247

Table 6.2: Execution time(s) for variable number of sub-platform and 2 workers.

The final set of experiments aimed to examine the impact of the encryption mechanism on the system's performance. We tested the runtime both with and without encryption, and the results are presented in table 6.3 and Fig. 6.5. The results indicate that the encryption mechanism does increase the execution time slightly, but the impact is very small. Sometimes, the effect is even smaller than the data fluctuations. This outcome is not surprising given that the amount of data requiring encryption in this system is relatively small, so the additional computation required is not significant.

WITHENCRYPTION	2P2W0C	2P3W0C	2P4W0C
TIME TAKEN	7.316	9.034	9.744
NOENCRYPTION	2P2W0C	2P3W0C	2P4W0C
TIME TAKEN	7.191	9.142	9.477

Table 6.3: Execution time(s) for variable number of sub-platform and 2 workers.

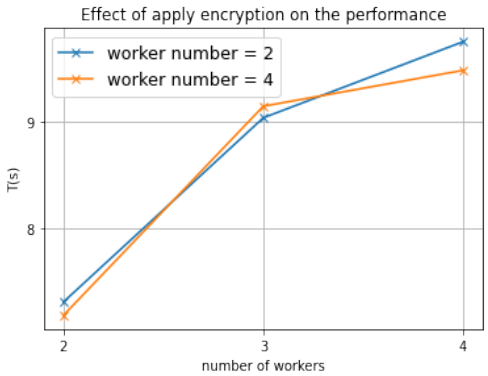


Figure 6.5: Performance graph of experiment of encryption mechanism's influence

Chapter 7

Discussion

This project implemented a hierarchical crowdsourcing system named ”PCS” based on Corda blockchain with privacy protection. In this chapter, I will discuss the project's achievements and its limitations.

One significant achievement of this project is the successful integration of Corda blockchain technology and a hierarchical crowdsourcing system to create a functional Crowdsourcing CorDapp that utilises blockchain technology to ensure secure and transparent data transmission with privacy protection. The use of Corda blockchain technology ensures that data is tamper-proof and immutable, thereby improving the system's security. Furthermore, unlike other famous public blockchain (e.g. Ethereum), Corda's property of permissioned blockchain ensures that only a select group of people can view a transaction, which enhances user privacy. The hierarchical structure of the system also facilitated efficient task distribution, which led to high-quality performance.

Besides that, PCS uses Paillier homomorphic encryption in the stage of task result evaluation and reward calculation, which ensures that some private data's details cannot be seen by data recipients who need to use this data for evaluation. This provides enhanced privacy, adding an additional layer of protection to the system. In addition, the current implementation of PCS follows a simulated real-world working scenario, and can be easily modified to suit the requirements of other actual scenarios.

However, there are some limitations of this project. One limitation of the system is that participants are required to possess knowledge of Corda blockchain technology and proficiency in using a CorDapp to take part in the system. However, the majority of ordinary users lack familiarity with blockchain technology, particularly on a notary blockchain platform such as Corda. Therefore, it may be difficult to quickly onboard users who are not familiar with the technology.

Another limitation is that PCS's current workflow is relatively abstract since it is a simulated realistic scenario, it may require further development to handle more complex tasks. Besides that, the tests carried out on this system are not sufficient. Due to limitations of the Corda local testing environment, the experiments on the operational performance of the system were not conducted with a large number of nodes involved. Therefore, future work could include conducting concurrency tests and large-scale stress

tests to assess the system's stability.

Overall, the project's achievements are significant, with the successful implementation of a hierarchical crowdsourcing system based on Corda blockchain with privacy protection. Nonetheless, the system still has some limitations that require addressing, and future work is needed to enable PCS's ongoing evolution and ensure its ability to meet the evolving needs of its users.

Chapter 8

Conclusion and Future Work

The object of our project is to implement a crowdsourcing services that reduces the traditional crowdsourcing system's issues. Here, we analyse the problems of traditional crowdsourcing systems and developed PCS based on these problems. PCS is a privacy preserved crowdsourcing service implemented on a notary-based permissioned blockchain platform called "Corda".

By utilising a decentralised blockchain platform to implement its service, PCS effectively mitigates the single point of failure problem inherent in traditional crowdsourcing systems. Its smart contract-based approach to service delivery also enhances transparency of evaluation and minimises reliance on subjective judgement from the main platform, which is a common issue with traditional systems. Additionally, by choosing the private blockchain Corda for development to minimise the number of observers for each transaction, and by using Paillier homomorphic encryption for the details of task results, the PCS system provides robust protection for the privacy of participants' information.

The project provides a detailed description of the PCS implementation code, along with comprehensive experiments that evaluate its actual operational performance specific to each flow. The PCS system was designed with a hierarchical structure, and developers can easily customise the system according to their specific requirements with minimal code modifications.

For future work, as discussed in Chapter 7, the current development scenario of the system is a simulated realistic scenario, and the service implementation is relatively abstract. Moving forward, it would be advisable to identify a more concrete usage scenario, pinpoint potential customers, make necessary system modifications, and release a Cordapp that can be directly used in production activities. Additionally, considering that regular users may lack knowledge of Corda blockchain, a user tutorial on the use of the system could be developed to facilitate users' quick understanding of how to operate the system. Furthermore, due to limitations in the Corda local testing environment, it was not possible to conduct testing with a large scale of participant nodes. Thus, future work could involve carrying out extra concurrency tests and stress tests to better assess PCS's stability.

Bibliography

[1] Corda. https://www.oreilly.com/library/view/blockchain-for-enterprise/9781788479745/f81fc099-fce7-4cdd-be92-32a4c5c48bc8.xhtml. Online;Accessed: 2023-3-17.

[2] Corda community edition key concepts. https://docs.r3.com/en/platform/corda/4.10/community/about-corda/corda-key-concepts.html. Online;Accessed: 2023-3-17.

[3] cordapp-template-java. https://github.com/corda/cordapp-template-java. Online;Accessed: 2023-3-17.

[4] Crowdsourcing: A definition. https://crowdsourcing.typepad.com/cs/2006/06/crowdsourcing_a.html. Online;Accessed: 2023-02-27.

[5] Elance and odesk hit by major ddos attacks, downing services for many freelancers. https://old.gigaom.com/2014/03/18/elance-hit-by-major-ddos-attack-downing-service-for-many-freelancers/. Online;Accessed: 2023-2-26.

[6] Flows. https://docs.r3.com/en/platform/corda/4.10/community/key-concepts-flows.html. Online;Accessed: 2023-3-18.

[7] Netflix spilled your brokeback mountain secret, lawsuit claims. https://www.wired.com/2009/12/netflix-privacy-lawsuit/. Online;Accessed: 2023-2-24.

[8] Notary service overview. https://docs.r3.com/en/platform/corda/5.0-beta/developing/ledger/notaries.html. Online;Accessed: 2023-3-17.

[9] These real-world data breach examples will make you rethink your data strategy. https://www.doherty.co.uk/blog/data-breach-examples-rethink-your-data-strategy/. Online;Accessed: 2023-02-27.

[10] Tokens sdk. https://docs.r3.com/en/platform/corda/4.10/community/token-sdk-introduction.html. Online;Accessed: 2023-3-17.

[11] Uber china statement on service outage on april 17. https://weibo.com/p/1001603832763923972414. Online;Accessed: 2023-2-26.

[12] What is a corda node. `https://docs.r3.com/en/platform/corda/4.8/enterprise/node/component-topology.html`. Online;Accessed: 2023-3-17.

[13] Shifa Manaruliesya Anggriane, Surya Michrandi Nasution, and Fairuz Azmi. Advanced e-voting system using paillier homomorphic encryption algorithm. In *2016 International Conference on Informatics and Computing (ICIC)*, pages 338–342, 2016.

[14] Federico Ast and Alejandro Sewrjugin. The crowdjury, a crowdsourced justice system for the collaboration era. *Crowdjury. org*, 1(9):1689–1699, 2015.

[15] Nazanin Zahed Benisi, Mehdi Aminian, and Bahman Javadi. Blockchain-based decentralized storage networks: A survey. *Journal of Network and Computer Applications*, 162:102656, 2020.

[16] Richard Gendal Brown. The corda platform: An introduction. *Retrieved*, 27:2018, 2018.

[17] Richard Gendal Brown, James Carlyle, Ian Grigg, and Mike Hearn. Corda: an introduction. *R3 CEV, August*, 1(15):14, 2016.

[18] Man Hon Cheung, Fen Hou, Jianwei Huang, and Richard Southwell. Distributed time-sensitive task selection in mobile crowdsensing. *IEEE Transactions on Mobile Computing*, 20(6):2172–2185, 2021.

[19] Usman W Chohan. The double spending problem and cryptocurrencies. *Available at SSRN 3090174*, 2021.

[20] Usman W Chohan. The double spending problem and cryptocurrencies. *Available at SSRN 3090174*, 2021.

[21] Kevin Dooley. *Designing large scale lans: Help for network designers*. " O'Reilly Media, Inc.", 2001.

[22] Mahdi Ghadamyari and Saeed Samet. Privacy-preserving statistical analysis of health data using paillier homomorphic encryption and permissioned blockchain. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 5474–5479, 2019.

[23] Mike Hearn and Richard Gendal Brown. Corda: A distributed ledger. *Corda Technical White Paper*, 2016, 2016.

[24] Jeff Howe et al. The rise of crowdsourcing. *Wired magazine*, 14(6):1–4, 2006.

[25] Viktor Jacynycz, Adrian Calvo, Samer Hassan, and Antonio A Sánchez-Ruiz. Betfunding: A distributed bounty-based crowdfunding platform over ethereum. In *Distributed computing and artificial intelligence, 13th international conference*, pages 403–411. Springer, 2016.

[26] Thivya Kandappu, Arik Friedman, Vijay Sivaraman, and Roksana Boreli. Privacy in crowdsourced platforms. *Privacy in a Digital, Networked World: Technologies, Implications and Solutions*, pages 57–84, 2015.

[27] Ming Li, Jian Weng, Anjia Yang, Wei Lu, Yue Zhang, Lin Hou, Jia-Nan Liu, Yang Xiang, and Robert H Deng. Crowdbc: A blockchain-based decentralized framework for crowdsourcing. *IEEE Transactions on Parallel and Distributed Systems*, 30(6):1251–1266, 2018.

[28] Roberto Llorente and Maria Morant. Crowdsourcing in higher education. *Advances in crowdsourcing*, pages 87–95, 2015.

[29] manishkk and kunerd. jpaillier. `https://github.com/kunerd/jpaillier`, 2018.

[30] Bhabendu Kumar Mohanta, Soumyashree S Panda, and Debasish Jena. An overview of smart contract and use cases in blockchain technology. In *2018 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, pages 1–4, 2018.

[31] Debajani Mohanty and Debajani Mohanty. Corda architecture. *R3 Corda for Architects and Developers: With Case Studies in Finance, Insurance, Healthcare, Travel, Telecom, and Agriculture*, pages 49–60, 2019.

[32] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized business review*, page 21260, 2008.

[33] Jeroen Oskam and Albert Boswijk. Airbnb: the future of networked hospitality businesses. *Journal of tourism futures*, 2(1):22–42, 2016.

[34] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Advances in Cryptology—EUROCRYPT'99: International Conference on the Theory and Application of Cryptographic Techniques Prague, Czech Republic, May 2–6, 1999 Proceedings 18*, pages 223–238. Springer, 1999.

[35] Peter E Rossi, Robert E McCulloch, and Greg M Allenby. The value of purchase history data in target marketing. *Marketing Science*, 15(4):321–340, 1996.

[36] Daniel J Solove and Danielle Keats Citron. Risk and anxiety: A theory of data-breach harms. *Tex. L. Rev.*, 96:737, 2017.

[37] Krushang Sonar and Hardik Upadhyay. A survey: Ddos attack on internet of things. *International Journal of Engineering Research and Development*, 10(11):58–63, 2014.

[38] Nick Szabo. Formalizing and securing relationships on public networks. *First monday*, 1997.

[39] Janice Warner. Business applications of crowdsourcing. *Proceedings of the Northeast Business & Economics Association*, 2011.

[40] Kerri Wazny. Applications of crowdsourcing in health: an overview. *Journal of global health*, 8(1), 2018.

[41] Wikipedia. *Wikipedia*. PediaPress, 2004.

[42] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[43] Huichuan Xia and Brian McKernan. Privacy in crowdsourcing: a review of the threats and challenges. *Computer Supported Cooperative Work (CSCW)*, 29:263–301, 2020.

[44] Panlong Yang, Qingyu Li, Yubo Yan, Xiang-Yang Li, Yan Xiong, Baowei Wang, and Xingming Sun. "friend is treasure": Exploring and exploiting mobile social contacts for efficient task offloading. *IEEE Transactions on Vehicular Technology*, 65(7):5485–5496, 2016.

[45] Xiang Zhang, Guoliang Xue, Ruozhou Yu, Dejun Yang, and Jian Tang. Keep your promise: Mechanism design against free-riding and false-reporting in crowdsourcing. *IEEE Internet of Things Journal*, 2(6):562–572, 2015.

Appendix A

Screenshots of Flow Executions

```
Tue Apr 11 00:16:23 BST 2023>>> start sendToken id: 17d05789-291f-469e-bda8-38f5085e7b14, amount: 4, receiver: platform
Starting
Done
Flow completed with result:
Transfer ownership of a token (token serial#: 17d05789-291f-469e-bda8-38f5085e7b14) to O=platform, L=London, C=GB
Transaction IDs: AAED5FAF944B6E0C648691AF8E7577673F834876D39C18237C2BA321FDF75970
 time taken: 0.936407301
```

Figure A.1: SendTokenFlow

```
Tue Apr 11 00:16:32 BST 2023>>> start requestFlow task: "valid", tokenId: 17d05789-291f-469e-bda8-38f5085e7b14
Starting
Collecting signatures from counterparties.
Verifying collected signatures.
Starting
Broadcasting transaction to participants
Done
Flow completed with result: request success, task id is: e4408eb8-d285-4b7f-9925-e548649c442d
 total time taken is: 0.2632418
```

Figure A.2: SendRequestFlow

```
start spread id: e4408eb8-d285-4b7f-9925-e548649c442d
valid
query success
Starting
task1: 8755d4d7-dd50-494a-98fa-67e49a4032fc
 task2: caa28ac8-63bb-4828-9967-b9101bb9b7e9
time taken: 0.0882268
Broadcasting transaction to participants
Done
Flow completed with result: SignedTransaction(id=42702F9DF326C5308AD36A4CECC04C6C5FB99FE29A722116EBE83D6CA2C7EB4A)
```

Figure A.3: SplitTaskFlow

```
Tue Apr 11 00:19:00 BST 2023>>> start acceptTask id: caa28ac8-63bb-4828-9967-b9101bb9b7e9
query success
Starting
verify Success
transaction created
Collecting signatures from counterparties.
Verifying collected signatures.
sessions added
totol time taken is: 1.087395001
Starting
Requesting signature by notary service
Requesting signature by Notary service
Validating response from Notary service
Broadcasting transaction to participants
Done
Flow completed with result: SignedTransaction(id=7F2B1C220737C459837E4DB8F9B6900493A9C8350103ED21782D490EBE4E05B8)
```

Figure A.4: AcceptTaskFlow

```
Tue Apr 11 00:19:31 BST 2023>>> start assignTask receivers: ["worker1_sup1","worker2_sup1","worker3_sup1"], taskId: caa2
8ac8-63bb-4828-9967-b9101bb9b7e9
Starting
query success
Broadcasting transaction to participants
Done
Flow completed with result: assign success.
task 1 is: 502c95a2-7ef1-410d-8660-0714f5e367fa
task2 is: a3949ede-91e1-459a-b0be-1723f09bf714
 total time taken is: 0.2177425
```

Figure A.5: AssignWorkFlow

```
start acceptWork id: 502c95a2-7ef1-410d-8660-0714f5e367fa, receiver: supplier2
here
query success
verify Success
transaction created
Starting
Collecting signatures from counterparties.
sessions added
total time taken is: 0.5750946
Verifying collected signatures.
Starting
Requesting signature by notary service
Requesting signature by Notary service
Validating response from Notary service
Broadcasting transaction to participants
Done
Flow completed with result: SignedTransaction(id=D86051360D9BD314882E6B9F0DE38B4C1CC978F833906EAAC04EACA52F13734F)
```

Figure A.6: AcceptWorkFlow

```
Tue Apr 11 00:20:39 BST 2023>>> start returnResult receiver: supplier2, id: 502c95a2-7ef1-410d-8660-0714f5e367fa, result
1: 80, result2: 80, key_name: key2
query success
Starting
Collecting signatures from counterparties.
sessions added
Verifying collected signatures.
Starting
Requesting signature by notary service
Requesting signature by Notary service
Validating response from Notary service
Broadcasting transaction to participants
Done
Flow completed with result: success, the result is: 176041931590084469278062122046452939172209058017935740152253888604178625358754339774460279650275573902762905774476453210838562941860177403297068082629998434108972216702177889345820486241374565449560910095833358457609612182913705894548877438111754143100080534010113900197615614981240872623225104198296317420179623687116814799409398424042628936642479768311960560615416902632036627675622912746593455955979689874856054954060184179943708699980224335913236206716664171860245377675473733255433303004431418664539110706622727166275415179172059412299533170911873814707120565272745184826062669113263032244995932101798002880098 6
time taken: 0.6939825
```

Figure A.7: ReturnWorkResultFlow

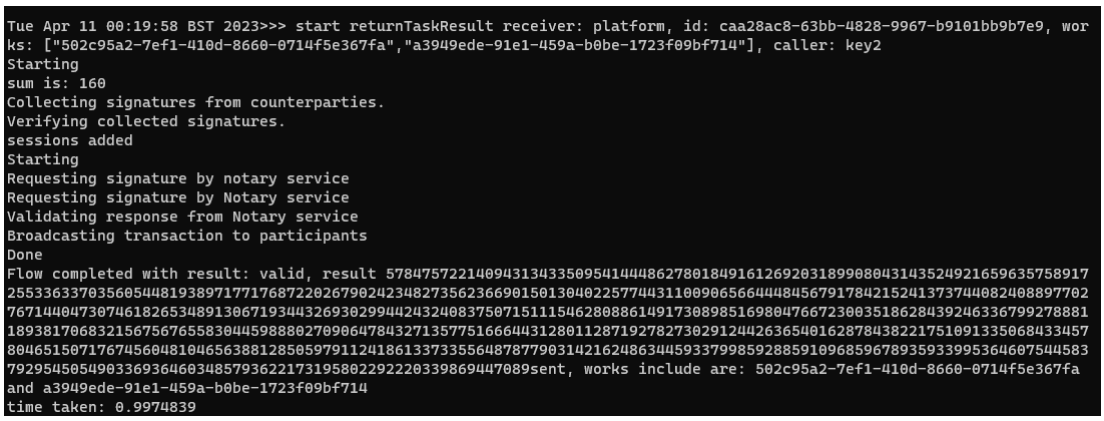


Figure A.8: ReturnResultFlow

Figure A.9: EvaluateFlow

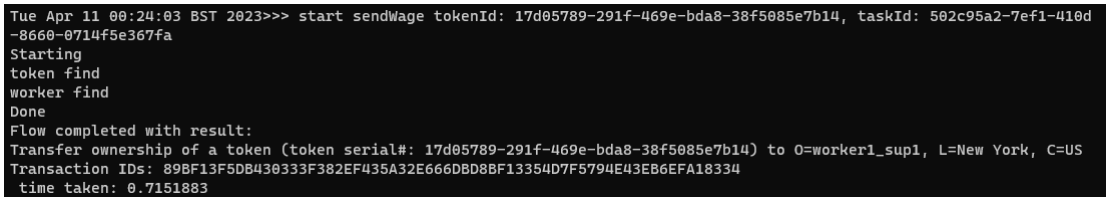


Figure A.10: SendWageFlow