Project Report

ZENG Jianyuan, ZHENG Qinyuan, YE Yuxiang

1. Abstract

As industries develop, the production of a product requires the support of many upstream industries. Therefore, the traceability information of a product will involve many industrial chains. Locating product information efficiently and accurately is a problem that needs to be solved. Blockchain technology is evolving. Compared to traditional traceability methods, it does not rely on a centralised database. In addition to its natural traceability, blockchain technology has advantages such as immutable and non-deletable. The use of blockchain technology for traceability will become a future trend.

However, the use of blockchain platforms is not uniform in different application scenarios and enterprise environments. To ensure the openness and transparency of data, some enterprises will use public chains to guarantee that data can be monitored. To maintain data privacy, other enterprises use federated chains to regulate access to nodes. If these two types of enterprises are engaged in commercial cooperation, how to achieve data interaction between the two platforms will become a problem to be solved as the data flow of the enterprises are on their respective chains.

In this project, we simulated the environment of car production. Firstly, data transfer was implemented on the Ethereum and Hyperledger Fabric platforms. Add, delete and change operations can be performed on car products and parts on these two platforms. This enables product traceability for a single enterprise on its blockchain. Secondly, we achieve interoperability of data within the two platforms mentioned above. Authorised users can access each other's data, enabling the interaction of car products and car part information. At the same time, we have designed the user interface. Users who have logged in can view and operate the pages according to their permissions. In our design, the user can locate the car product and all the parts included in this product, enabling accurate and comprehensive product traceability.

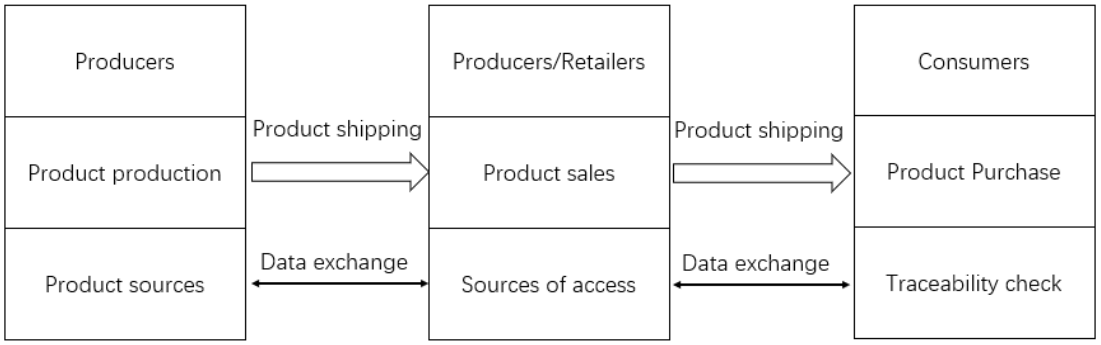
2. Introduction

A blockchain can be thought of as a decentralised and tamper-proof distributed database. Each node in the blockchain system replicates and backs up the ledger synchronously to ensure that the blockchain is fully decentralised and that the data is open, reliable and permanent. These features of blockchain provide a new solution to achieve reliable traceability.

The inherent traceability of blockchain is an excellent match to the needs of the current traceability scenario in the supply chain. The data recorded on the blockchain is open and transparent between the parties to the transaction. Product flow data in the supply chain can be corresponded to the blockchain, creating a complete and trusted flow of information. Keeping information in the block ensures that the information source of the system is unique

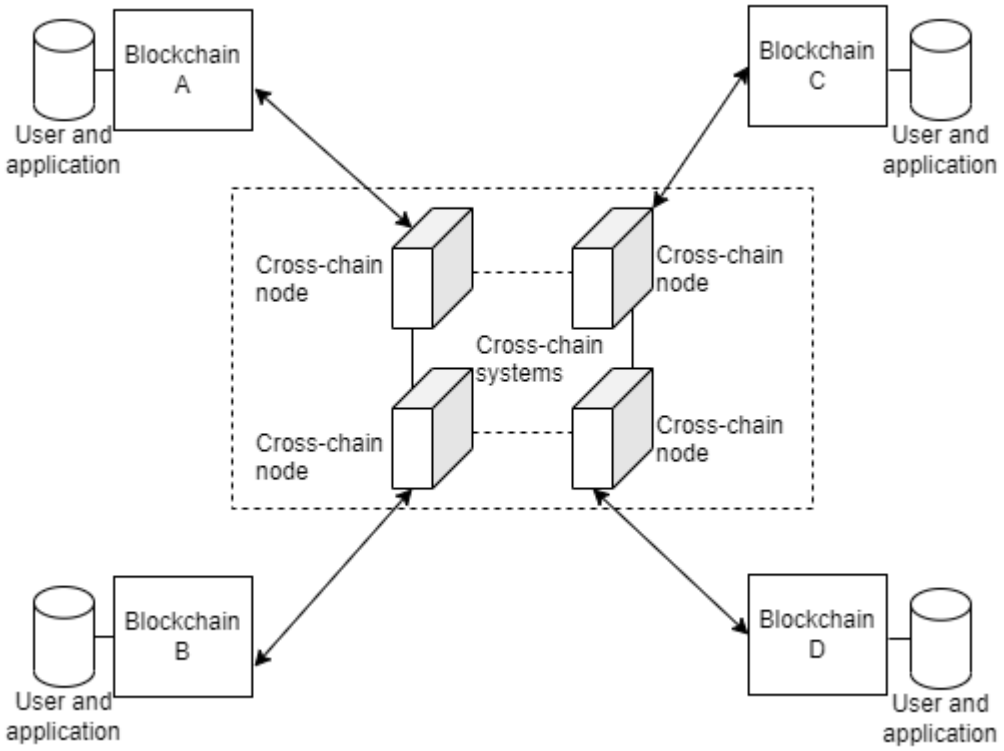
and reliable, and data tracing facilitates the system to detect counterfeiting. At the same time, the smart contract on the blockchain is automatically triggered, recorded, and supervised, which can effectively remove the influence of human operation on the data.

The following diagram illustrates the blockchain traceability model.



In practice, the traceability of a product will involve many types of industries. Moreover, each industry will maintain a traceability chain of its own. Interaction between product chains is necessary in order to enable users to trace product information in its entirety. A clear upstream and downstream relationship of parts throughout the product's lifecycle will make traceability information more reliable. Therefore, there is an excellent need for cross-chaining of blockchains.

Cross-chaining is a technology to allow value to cross the barriers between chains. This allows information or resources originally on one chain to be converted to information or resources on another chain, thus enabling the circulation of value. Unlike the Internet's TCP/IP communication protocol, it is only necessary to ensure that the party receives the message successfully. Blockchain cross-chains do not have the ability to send repeatedly after a transmission has been interfered with by others. Because the data between ledgers needs to be kept in sync, consistent changes to the ledger are required. Otherwise, value loss or double spending problems may occur. Therefore, cross-chaining is more than just the transfer of information; it is essentially the process of value flowing between different blockchains under the premise of value conservation.



The diagram above shows the basic architecture of cross-chain system interaction. The actors involved include each independent blockchain as well as their users and applications. Each parallel blockchain interacts with data via cross-chain nodes. This allows users on the chain to access information located on other chains within their rights.

Cross-chain technology for blockchains is an important method for blockchains to achieve interoperability and improve scalability. It addresses the need for data interaction between two or more blockchains. At the same time, it breaks the blockchain data silos, helps the chain of different enterprises to be mutually trustworthy and linkable, and promotes the credible integration of the industrial ecology on the traceability chain.

In the automotive industry, for example, a brand may have several production lines which have their own traceability chains. In order for brands to be able to plan their production lines in an integrated manner, these lines need to exchange information with each other in order to improve optimisation. So, achieving interaction across multiple chains is a great boost to today's industry. This approach also increases the efficiency of business enquiries and makes the industry more flexible and scalable.

3. Related Work

The development of blockchain technology and the economy has created a demand for blockchain cross-chaining. Implementing blockchain cross-chaining not only maintains the decentralised concept of blockchain but also significantly improves the transaction performance of the blockchain network. With the development and refinement of smart contract development platforms, a large number of vertical public chains and blockchains

applied to the business sector have emerged. These chains form numerous independent infrastructures and business systems. At the same time, as they develop, these blockchains all need to expand the diversity of their application functions to enable interconnection between multiple chains. Thus, the inter-chain flow of value and business. Cross-chain technology is a bridge for blockchain outward expansion and inter-chain interaction and an important technology for enabling data sharing between blockchain networks(Lin et al., 2021).

According to the scope of cross-chain execution, cross-chains can be classified into homogeneous cross-chains and heterogeneous cross-chains. Homogeneous chains refer to a type of blockchain where the block structure, consensus method, network architecture, data security assurance and smart contracts are all the same, so cross-chain interaction between homogeneous chains is easy to achieve. However, most of the blockchain projects in real-life applications come from different industries and fields. They are heterogeneous chains developed by different enterprise teams using different technical architectures based on different scenario requirements and design concepts. The structure and execution logic of each chain differ greatly, which increases the difficulty of cross-chain interaction between heterogeneous chains, and this is the main challenge to be solved in the cross-chain field.

Nolan first introduced the idea of atomic transfer in 2013. The atomic transfer is the idea that transactions on both sides of a cross-chain either happen at the same time or remain static at the same time and that the two are inseparable. It was one of the first technical solutions used for cross-chain transactions on the blockchain, which was eventually refined and adapted into the hash-locking technique(Monika & Bhatia, 2020). Poon has released a white paper on the Lightning Network. Introducing the Lightning Network (Lee & Kim, 2020) into the Bitcoin architecture enhances the throughput and transaction processing rate of the Bitcoin network. It improves the transaction channel under the Bitcoin chain and solves the problem of confirming off-chain transactions. Anchored sidechains (Singh et al., 2020) make use of a two-way anchoring mechanism. It enables the transfer of crypto assets on the main chain and the sidechain at a certain exchange rate. Relay chains (Westerkamp & Eberhardt, 2020) were first used for one-way cross-chain communication from the Bitcoin platform to the Ethernet platform, and its use solves the problem of cross-chain communication between blockchains on different platforms, further extending the ability to interact across chains.

Babu Pillai (Pillai et al., 2020)et al. propose a cross-communication model that works at the application layer to address cross-communication between blockchain-based systems without the need for intermediaries. This process also ensures the authenticity of the generated messages and does not change the heterogeneous nature of the blockchain system. Compared to the relay chain model, this approach saves time and reduces overhead by eliminating the process of entering messages into the relay chain. Victor Zakhary(Zakhary et al., 2020) et al. note that in hash time-locking protocols, time-lock expiration can lead to violations of all-or-nothing atomicity. An honest participant may end up losing assets if a smart contract is not executed on time due to a crashing failure of its site or network delays. Therefore, the paper proposes a decentralised all-or-nothing atomic cross-chain commitment protocol that ensures the atomicity of transactions even in the event of failure. Polkadot(Kan et al., 2018)、Cosmos(Burdges J et al., 2020) cross-chain platform first implements a cross-chain mechanism for relay chains. Users can be compatible with various blockchain applications through cross-chain platforms. However, these platforms do

not have a more systematic compatibility standard in terms of cross-chain functionality and cross-chain, and the compatibility is not strong enough.

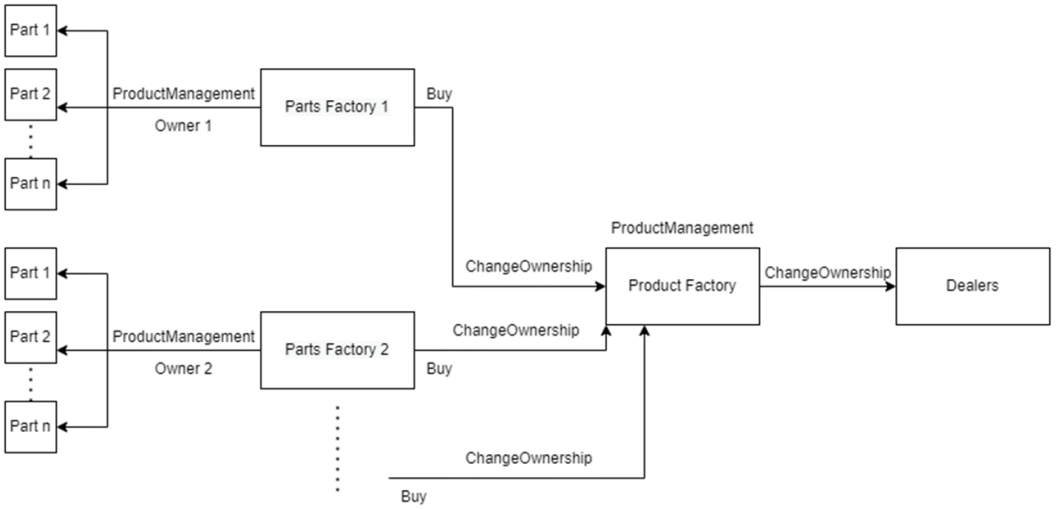
In 2016, Vitalik Buterin, the founder of Ethernet, summarised in the literature hash locking, notary mechanisms and sidechain/relay technologies. These techniques, together with the later proposed distributed private key control techniques, are the four dominant techniques for implementing blockchain cross-chains. Most of the existing cross-chain projects are developed based on these four techniques.

4. System design

4.1 Ethereum

4.1.1 Structure design

The above diagram shows the basic structure design of the traceability chain of the Ethernet



platform.

The overall flow of the traceability chain is as follows: Parts Factory produces the parts, Product Factory assembles the parts, and then Dealers perform the purchase operation. At each step of the process, information about the product or part is recorded and saved, and can be viewed at the same time, thus achieving the purpose of traceability.

The system consists of three main roles: Parts Factory, Product Factory and Dealers.

1. The Parts Factory is responsible for the production of parts for products. Each factory registers the parts it produces. In this case, the Parts Factory is the owner of the part.

2. The Product Factory is responsible for manufacturing products using parts made by Parts Factory. For example, part1 (tyres) from Parts Factory1 (tyre manufacturer),

parts2 (doors) from Parts Factory2 (door manufacturer), Etc. are used to manufacture a car. In this process, the factory also registers the product, and the owner of the product is then marked as Product Factory.

3. The Dealer is in the role of purchasing goods from the Product Factory. At the time of purchase, the ownership of the product changes. The Owner of the product changes from a Product Factory to a Dealer.

During the use of the traceability chain system, the product undergoes the following four main operations.

1. Registration of parts. At Parts Factory, products that have gone through the production process will be marked. A map will be created to give the details of the part itself (part type, serial number and manufacturing date), the factory that made it (factory id) and the current owner (owner id)。

2. ProductManagement. The manufacturer buys the parts from Parts Factory and produces the complete product. A map will be created to give the details of the parts used for this product and to register the current product.

3. ChangeOwnership. This operation exists in two main steps: firstly, when Parts Factory is producing the part, it marks the owner of the part with this operation; secondly, when Product Factory purchases the part for production, it changes the owner of the part to Product Factory.

4. Check Products and Parts. In the supply chain, users can look up current and historical information on parts and products.

The traceability chain implements the query operation as follows.

First, we have the parts factory, responsible for producing different parts of a product. The factory informs each part production using the “ProductManagement” smart contract, which will keep details about each part and product. Additionally, the factory states that it is the owner of a specific part by calling a method from the “ChangeOwnership” smart contract, which will keep the owner history of each part and product.

Going further on the chain, we have a production factory that buys parts from the part factory to manufacture their product. The “ChangeOwnership” contract is where we keep this operation, so we have another method to allow the parts factory to transfer the part ownership to the car factory. With enough parts, the factory can finally start to manufacture their products. Similar to what the parts factory did to inform its job, the car factory now uses the “ProductManagement” smart contract to state a specific car assembly. Each product has a set of properties, like a serial number, and a parts list, that ties it to specific parts. The ownership is controlled by the “ChangeOwnership” contract, and so the factory sets that product ownership to itself.

Finally, we add dealers to the scenario and they can buy the product from a factory and the latter can transfer the product ownership using “ChangeOwnership”. The blockchain stores every transaction that involved that product or any of the parts that compose it, so the dealer

(or any other part) can track everything that a specific item has gone through. In our case this tracking will be done by watching the events generated by the transactions.

4.1.2 Code details

Structure definition as follows.

```
struct Part{
    address manufacturer;
    string serial_number;
    string part_type;
    string creation_date;
}

struct Product{
    address manufacturer;
    string serial_number;
    string product_type;
    string creation_date;
    bytes32[6] parts;
}
```

```
mapping(bytes32 => Part) public parts;
mapping(uint => bytes32) public parts_list;
mapping(bytes32 => Product) public products;
mapping(uint => bytes32) public products_list;

uint public parts_cnt = 0;
uint public products_cnt = 0;
```

Both Part and Product contain the information of manufacture, serial_number, part_type and creation_date. In addition, in the Product section, we use an array to store the information about the parts. We have used map to create correspondences for querying.

In the contract of ProductManagement, we have four methods.

```
function concatenateInfoAndHash(address a1, string memory s1, string memory s2, string memory s3) private returns (bytes32){…
}


function buildPart(string memory serial_number, string memory part_type, string memory creation_date) public {…
}

function buildProduct(string memory serial_number, string memory product_type, string memory creation_date, bytes32[6] memory part_array)
}

function getParts(bytes32 product_hash) public returns (bytes32[6] memory){…
}
```

In the contract of ChangeOwnership, we have two methods.

```
contract ProductManagement{
    struct Part{…
    }

    struct Product{…
    }

    mapping(bytes32 => Part) public parts;
    mapping(bytes32 => Product) public products;

    function getParts(bytes32 product_hash) public returns (bytes32[6] memory) {}
}

contract ChangeOwnership {

    enum OperationType {PART, PRODUCT}
    mapping(bytes32 => address) public currentPartOwner;
    mapping(bytes32 => address) public currentProductOwner;
```

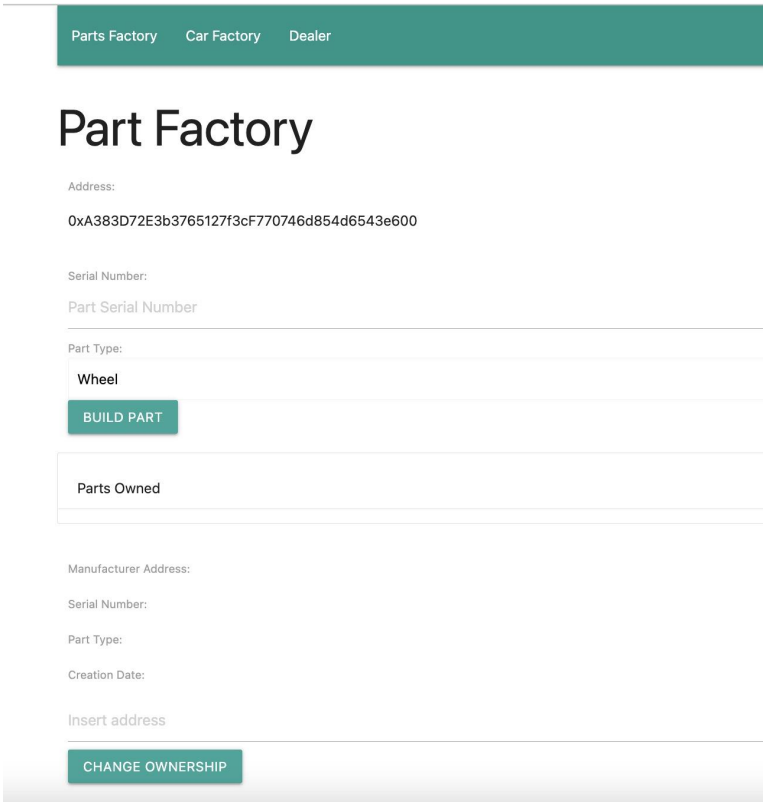
One is addOwnership.

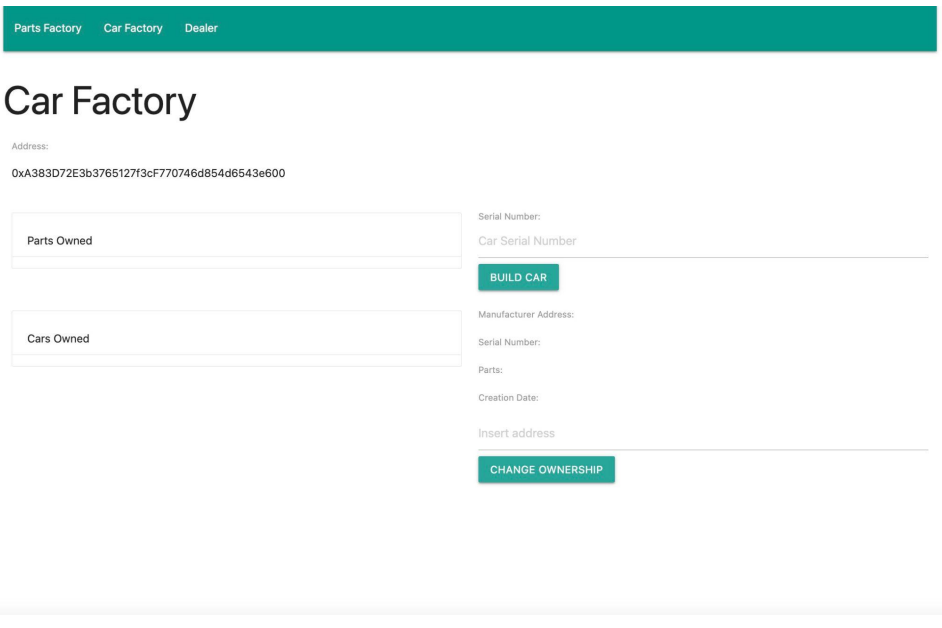
```
function addOwnership(uint op_type, bytes32 p_hash) public returns (bool) {
    if(op_type == uint(OperationType.PART)){
        address manufacturer;
        (manufacturer, , , ) = pm.parts(p_hash);
        require(currentPartOwner[p_hash] == address(0), "Part was already registered");
        require(manufacturer == msg.sender, "Part was not made by requester");
        currentPartOwner[p_hash] = msg.sender;
        emit TransferPartOwnership(p_hash, msg.sender);
    } else if (op_type == uint(OperationType.PRODUCT)){
        address manufacturer;
        (manufacturer, , , ) = pm.products(p_hash);
        require(currentProductOwner[p_hash] == address(0), "Product was already registered");
        require(manufacturer == msg.sender, "Product was not made by requester");
        currentProductOwner[p_hash] = msg.sender;
        emit TransferProductOwnership(p_hash, msg.sender);
    }
}
```

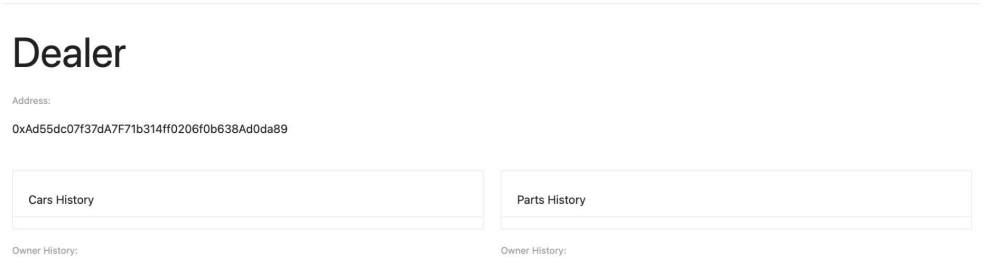
Another is changeOwnership.

```
function changeOwnership(uint op_type, bytes32 p_hash, address to) public returns (bool) {
  //Check if the element exists and belongs to the user requesting ownership change
    if(op_type == uint(OperationType.PART)){
        require(currentPartOwner[p_hash] == msg.sender, "Part is not owned by requester");
        currentPartOwner[p_hash] = to;
        emit TransferPartOwnership(p_hash, to);
    } else if (op_type == uint(OperationType.PRODUCT)){
        require(currentProductOwner[p_hash] == msg.sender, "Product is not owned by requester");
        currentProductOwner[p_hash] = to;
        emit TransferProductOwnership(p_hash, to);
        //Change part ownership too
        bytes32[6] memory part_list = pm.getParts(p_hash);
        for(uint i = 0; i < part_list.length; i++){
            currentPartOwner[part_list[i]] = to;
            emit TransferPartOwnership(part_list[i], to);
        }

    }
}
```

The implementation of Ethereum works as follows.

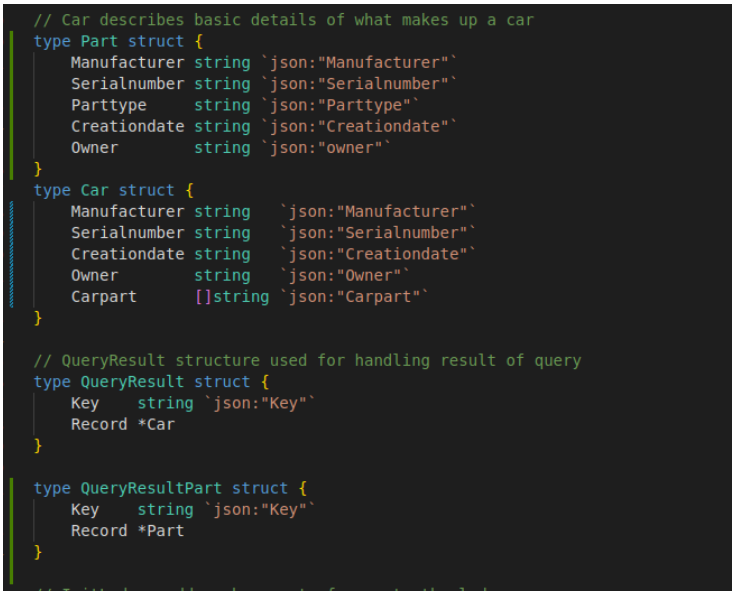

Parts Factory
Car Factory
Dealer
Part Factory
Address:
0xA383D72E3b3765127f3cF770746d854d6543e600
Serial Number:
Part Serial Number
Part Type:
Wheel
BUILD PART
Parts Owned
Manufacturer Address:
Serial Number:
Part Type:
Creation Date:
Insert address
CHANGE OWNERSHIP


Parts Factory
Car Factory
Dealer
Car Factory
Address:
0xA383D72E3b3765127f3cF770746d854d6543e600
Parts Owned
Cars Owned
Serial Number:
Car Serial Number
BUILD CAR
Manufacturer Address:
Serial Number:
Parts:
Creation Date:
Insert address
CHANGE OWNERSHIP


Dealer
Address:
0xAd55dc07f37dA7F71b314ff0206f0b638Ad0da89
Cars History
Owner History:
Parts History
Owner History:

4.2 Hyperledger Fabric

Hyperledger data strusct is shown below

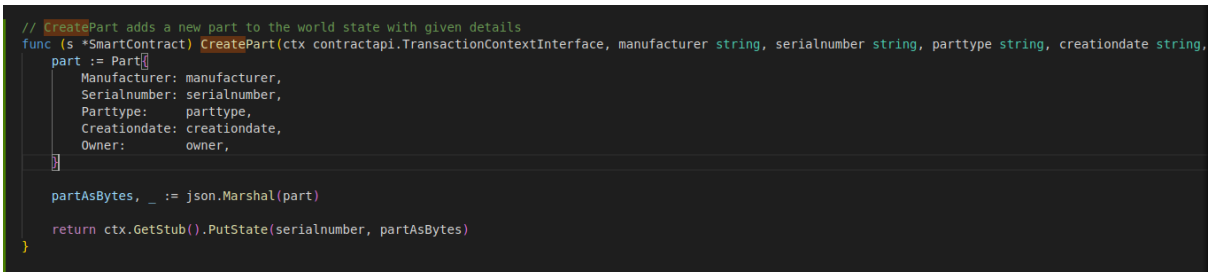


```go
// Car describes basic details of what makes up a car
type Part struct {
    Manufacturer string `json:"Manufacturer"`
    Serialnumber string `json:"Serialnumber"`
    Parttype     string `json:"Parttype"`
    Creationdate string `json:"Creationdate"`
    Owner        string `json:"owner"`
}
type Car struct {
    Manufacturer string   `json:"Manufacturer"`
    Serialnumber string   `json:"Serialnumber"`
    Creationdate string   `json:"Creationdate"`
    Owner        string   `json:"Owner"`
    Carpart      []string `json:"Carpart"`
}

// QueryResult structure used for handling result of query
type QueryResult struct {
    Key    string `json:"Key"`
    Record *Car
}

type QueryResultPart struct {
    Key    string `json:"Key"`
    Record *Part
}
```

Search existing part by owner / manufacturer:

```go
//search part by owner
func (s *SmartContract) Searchpart(ctx contractapi.TransactionContextInterface, keyword string) ([]QueryResultPart, error) {
    var query string
    // query = fmt.Sprintf("{\"selector\":{\"owner\": {\"$regex\":\"(?i)^.*?%s.*?$\"}}}", owner)
    query = fmt.Sprintf("{\"selector\":{\"$or\":[{\"manufacturer\":{\"$regex\":\"(?i)^.*?%s.*?$\"}},{\"owner\":{\"$regex\":\"(?i)^.*?%s.*?$\"}}]}}",keyword,key
    fmt.Printf(query)
    resultsIterator, err := ctx.GetStub().GetQueryResult(query)
    if err != nil {
        return nil, err
    }

    results := []QueryResultPart{}

    for resultsIterator.HasNext() {
        queryResponse, err := resultsIterator.Next()

        if err != nil {
            return nil, err
        }

        part := new(Part)
        _ = json.Unmarshal(queryResponse.Value, part)

        QueryResultPart := QueryResultPart{Key: queryResponse.Key, Record: part}
        results = append(results, QueryResultPart)
    }
    return results, nil
}
```

Create a new part record



```go
// CreatePart adds a new part to the world state with given details
func (s *SmartContract) CreatePart(ctx contractapi.TransactionContextInterface, manufacturer string, serialnumber string, parttype string, creationdate string,
    part := Part{
        Manufacturer: manufacturer,
        Serialnumber: serialnumber,
        Parttype:     parttype,
        Creationdate: creationdate,
        Owner:        owner,
    }

    partAsBytes, _ := json.Marshal(part)

    return ctx.GetStub().PutState(serialnumber, partAsBytes)
}
```

Create a new car record

```go
// CreateCar adds a new car to the world state with given details
func (s *SmartContract) CreateCar(ctx contractapi.TransactionContextInterface, manufacturer string, serialnumber string, creationdate string, owner string, par
    car := Car{
        Manufacturer: manufacturer,
        Serialnumber: serialnumber,
        Creationdate: creationdate,
        Owner:        owner,
    }

    for i, partnumber := range partserial {
        part, err := s.QueryPart(ctx, partnumber)
        if err != nil {
            return err
        }
        if part.Owner != manufacturer {
            return fmt.Errorf("Not Owner of the part")
        }
        car.Carpart[i] = partnumber
    }

    carAsBytes, _ := json.Marshal(car)

    return ctx.GetStub().PutState(serialnumber, carAsBytes)
}
```

Search the part by its serial number

```go
// QueryPart returns the part stored in the world state with given id
func (s *SmartContract) QueryPart(ctx contractapi.TransactionContextInterface, serialNumber string) (*Part, error) {
    partAsBytes, err := ctx.GetStub().GetState(serialNumber)

    if err != nil {
        return nil, fmt.Errorf("Failed to read from world state. %s", err.Error())
    }

    if partAsBytes == nil {
        return nil, fmt.Errorf("%s does not exist", serialNumber)
    }

    part := new(Part)
    _ = json.Unmarshal(partAsBytes, part)

    return part, nil
}
```

Change the car owner(Ship the product out)

```go
// ChangeCarOwner updates the owner field of car with given id in world state
func (s *SmartContract) ChangeCarOwner(ctx contractapi.TransactionContextInterface, carNumber string, newOwner string) error {
    car, err := s.QueryCar(ctx, carNumber)
    if err != nil {
        return err
    }
    car.Owner = newOwner

    for i := range car.Carpart {
        part, err := s.QueryPart(ctx, car.Carpart[i])
        if err != nil {
            return err
        }
        part.Owner = newOwner
        partAsBytes, _ := json.Marshal(part)
        ctx.GetStub().PutState(part.Serialnumber, partAsBytes)
    }
    carAsBytes, _ := json.Marshal(car)

    return ctx.GetStub().PutState(carNumber, carAsBytes)
}
```

Change the part owner

```go
// ChangePartOwner updates the owner field of part with given id in world state
func (s *SmartContract) ChangePartOwner(ctx contractapi.TransactionContextInterface, serialnumber string, newOwner string) error {
    part, err := s.QueryPart(ctx, serialnumber)

    if err != nil {
        return err
    }

    part.Owner = newOwner

    partAsBytes, _ := json.Marshal(part)

    return ctx.GetStub().PutState(serialnumber, partAsBytes)
}
```

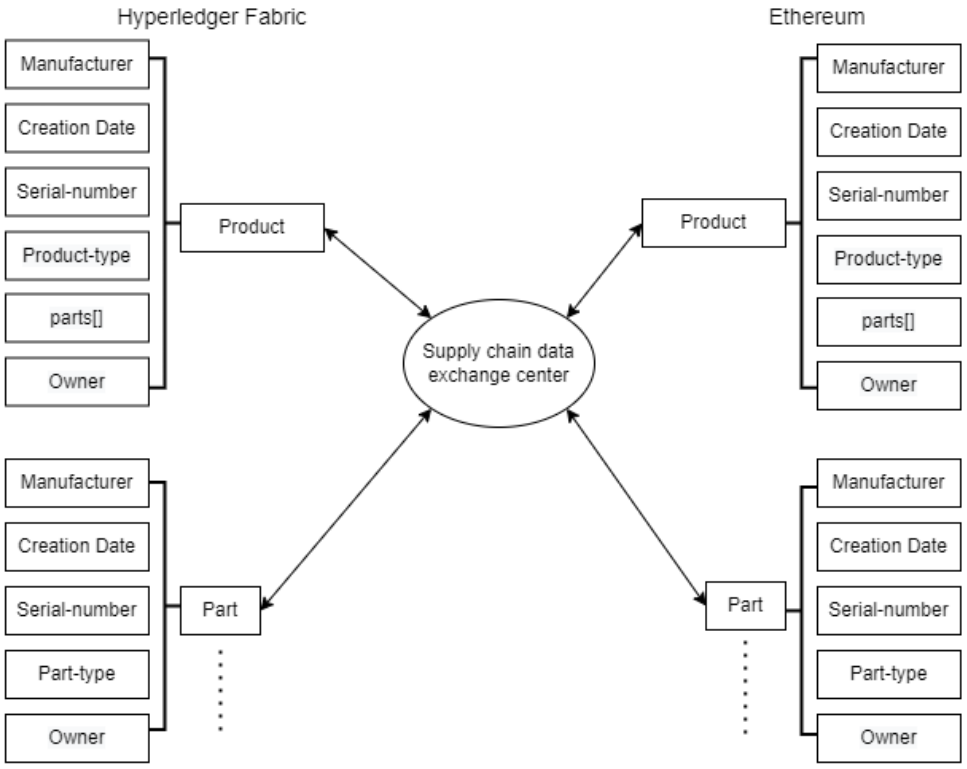
Record the data from Ethereum (assets exchange)

```go
func (s *SmartContract) GetDataFromOutside(ctx contractapi.TransactionContextInterface, serialnumber string, data string) {
    dataAsBytes := []byte(data)
    ctx.GetStub().PutState(serialnumber, dataAsBytes)
}
```

4.3 Design of cross-platform interaction

4.3.1 Structure design

We plan to implement the data interaction according to the following diagram.



We designed both the Ethereum platform and the Hyperledger Fabric platform to have the same Part and Product information. The data is exchanged through a data exchange centre. The center include all the contract methods of both blockchains. So,we can do Query and Make Assets operations to both Hyperledger and Ethereum on the exchange center.

The steps for requesting data interaction are as follows.

1. Query all the assets we have (by input clientOrgID and eth address).
2. We compare the assets data(part and product) stored in different blockchain and find the difference.
3. Then we can merge the store in different blockchains.

4.3.2 Code details

```
var express = require('express');
var router = express.Router();
const FabricClient = require('../fabric/client.js');
const client = new FabricClient();
var Web3 = require('web3');
var provider = 'http://127.0.0.1:8545';
var web3Provider = new Web3.providers.HttpProvider(provider);
var web3 = new Web3(web3Provider);

```

```
pm = new web3.eth.Contract(pmabi, pmaddress);

async function getContractPublicVariable() {
    var cnt = new Array();
    for(i=0;i<4;i++){
        var value = await pm.methods.parts_list(i).call()
        cnt[i] = value;
    }
    console.log(cnt);
    for(i=0;i<4;i++){
        var result = await pm.methods.parts(cnt[i]).call()
        console.log(result);
    }
}
getContractPublicVariable();
```

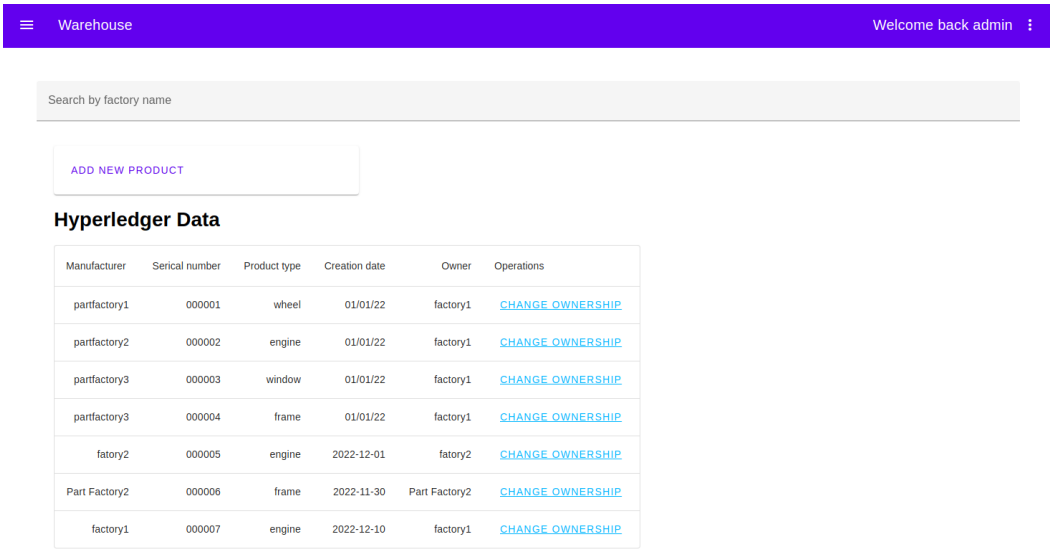
```
router.get('/', function(req, res, next) {
  var productRequest = new Promise((resolve, reject) => {
    if (req.query.search){
      console.log(req.query.search)
      var products = client.query(req.username, 'SearchProducts', [req.query.search]);
```

4.3.3 Front end Interface

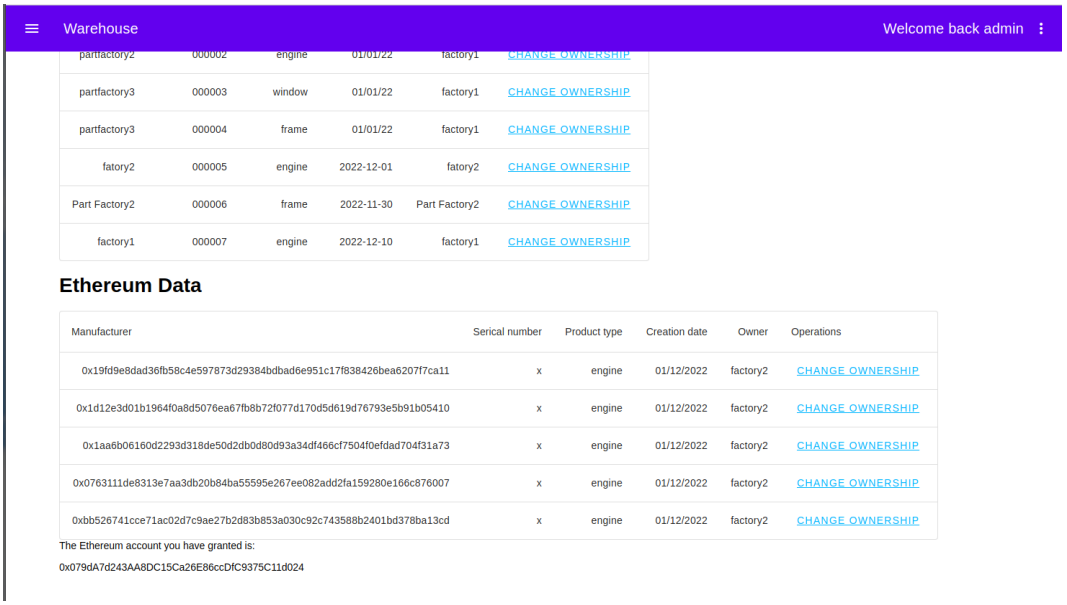
- Login
  - User need to log in with username and password to access the content

  -
  - System will reject login request if enter Incorrect username or password

- Index page
- Hyperledger Data, user can see all the product information store in the hyperledger fabric

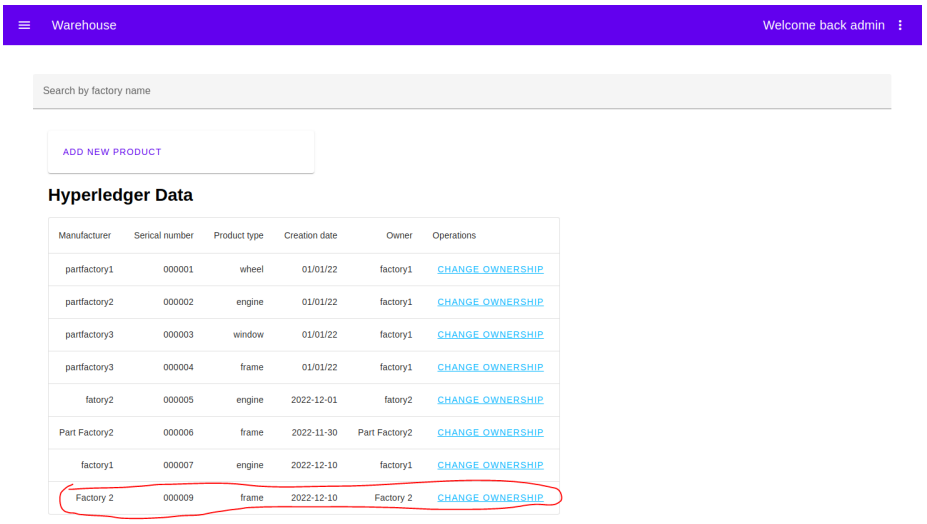


- Ethereum Data, user can access all the information store in the Ethereum blockchain



- Add new product button will direct the user to the creat product page, authority user(admin/appuser) can create a new product record

- If add new product successfully, a new record to append to the hyperledger fabric



5. Deployment guide

System requirement: Ubuntu 20.04.1 LTS
Truffle v5.6.8 (core: 5.6.8)
Ganache v7.5.0
Solidity v0.5.16 (solc-js)
Node v14.21.1
Web3.js v1.7.4
Npm version: 6.14.17
Go: go version go1.13.8 linux/amd64
Ethereum

1. Requirements:
    - Truffle
        - `npm install -g truffle`
    - Ganache-CLI
        - `npm install ganache --global`
2. Ethereum Testnet start up
    - `ganache`
3. Deploy smart contract
    - In “~/supplychain/SAB-ethereum-supply-chain” directory, run a truffle console , then run ”truffle migrate”

- If success, record the second contract address in the truffle console and put it into the

Hyperledger Fabric

1. Requirements:
    - fabric-samples folder: (Environement from Lab2 VM)
    - (from github or we upload the files to github)
2. Hyperledger Fabric testnet start up
    - In directory '~/fabcar2', run "./startFabric.sh"
3. Register user
    - In directory '~/fabcar2'/javascript'
    - After node installation or upgradtion, npm cache will be clean. We recommend to run "npm install fabric-ca-client" in ~/javascript to install the fabric SDK dependency for the nodejs action in the following section.
    - Run " node enrollAdmin.js"
    - Run "node registerUser.js"
4. Hyperledger Fabric testnet will initialise by the chain code with InitLedger() function.

Front-end user interface

Login info: Username: *admin* password:*root*

1. In directory "~/fabcar2'/javasript/myapp"
    - Run "npm install"
    - Run "npm start"
    - Open web brower and enter "localhost:3000" to access the web interface.

6. Evaluation and Conclusion

6.1 Evaluation

1. Security. In our system, user need to use their user name and password to log in to the system to access to the contents, if not authorities user attend to access or modify the information, he will be rejected to the login page.
2. Accessibility. In our system, only admin user can have privilege to do modification to the dataset, normal user can only access to the data. They can not change the ownership of the products/parts. Admin user is the certifaicated authority entity from the hyperledger fabric. Because in the Ethereum part we can not identify the user privilege from their accounts, so if the user from the Ethereum part want to modify the information store in the Hyperledger Fabric, he need to request a Hyperledger Fabric administer to add him as a admin user first to become to be able to do modification.
3. Efficiency. In our system, user in one company can easily access the product information from the other's. Our system act as a trusted third party data centre to help two manaufacters company using different blockchain technology trace product information.

6.2 Conclusion

In this project, we use the automotive industry as a background. A complete car product consists of many car parts. Product and part information is stored on different blockchain platforms. We implemented data upload and query on the Ethereum platform and on the Hyperledger Fabric platform, respectively. Users of both platforms can implement basic operations on the data on their respective platforms. In addition, we have implemented data interaction between the two platforms using JavaScript. After logging in, users can manipulate data on both platforms according to their rights. However, there is room for improvement in our design.

In future development, we can add a certificated contract to the Ethereum part to verify users in the manufacturer using Ethereum. This can improve the system's security and accessibility to have a fully trusted assets exchange between two entities. Also, the front-end interface needs to implement assets exchange functionality to help users from Hyperledger or Ethereum exchange the assets easily, rather than use the command line to perform product exchange.

Reference

Burdges, J., Cevallos, A., Czaban, P., Habermeier, R., Hosseini, S., Lama, F., ... & Wood, G. (2020). Overview of polkadot and its design considerations. arXiv preprint arXiv:2005.13456.

Kan, L., Wei, Y., Hafiz Muhammad, A., Siyuan, W., Gao, L. C., & Kai, H. (2018). A multiple blockchains architecture on inter-blockchain communication. *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. https://doi.org/10.1109/qrs-c.2018.00037

Lee, S., & Kim, H. (2020). On the robustness of lightning network in bitcoin. *Pervasive and Mobile Computing*, *61*, 101108. https://doi.org/10.1016/j.pmcj.2019.101108

Lin, S., Kong, Y., & Nie, S. (2021). Overview of block chain cross chain technology. *2021 13th International Conference on Measuring Technology and Mechatronics Automation (ICMTMA)*. https://doi.org/10.1109/icmtma52658.2021.00083

Monika, & Bhatia, R. (2020). Interoperability Solutions for Blockchain. *2020 International Conference on Smart Technologies in Computing, Electrical and Electronics (ICSTCEE)*. https://doi.org/10.1109/icstcee49637.2020.9277054

Pillai, B., Biswas, K., & Muthukkumarasamy, V. (2020). Cross-chain interoperability among blockchain-based systems using transactions. *The Knowledge Engineering Review*, *35*. https://doi.org/10.1017/s0269888920000314

Singh, A., Click, K., Parizi, R. M., Zhang, Q., Dehghantanha, A., & Choo, K.-K. R. (2020). Sidechain technologies in Blockchain Networks: An examination and

state-of-the-art review. *Journal of Network and Computer Applications*, *149*, 102471. https://doi.org/10.1016/j.jnca.2019.102471

Westerkamp, M., & Eberhardt, J. (2020). Zkrelay: Facilitating sidechains using zkSNARK-based chain-relays. *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. https://doi.org/10.1109/eurospw51379.2020.00058

Zakhary, V., Agrawal, D., & El Abbadi, A. (2020). Atomic commitment across blockchains. *Proceedings of the VLDB Endowment*, *13*(9), 1319–1331. https://doi.org/10.14778/3397230.3397231

Appendix

Project code access:
https://connectpolyu-my.sharepoint.com/:f:/g/personal/22045343g_connect_polyu_hk/Er4oBD1T5blMjXUh8wz8K2AB90K-uhZSWuAbKdZN4TbXMg?e=8dv2NE

Demo Video:
Part1: https://youtu.be/Q4zYupv73Gk

Part2:https://youtu.be/1cL6Q70O-JI