# Decentralized Container Tracking System

*Author:* Carlos Jácome Ferrao

*Director:* Manuel Esteve Domingo

*Start date:* 9/02/2020

*Objectives* — The objective of this thesis is the design of a decentralized application that allows not only tracking cargo containers in a single immutable ledger, but also to pay for the services incurred during their journey. As an additional objective, a prototype will be implemented as proof of concept and testing bench for further research.

*Methodology* — The methodology employed is, in the first place, the research of technical publications to understand the fundamental concepts behind blockchain technology. Subsequently, the design and implementation of a smart contract to handle the prototype business logic. Finally, the implementation of a user interface to allow final users to interact with the smart contract.

*Prototype* – A decentralized application has been developed over Ethereum blockchain platform to demonstrate the feasibility of the architecture proposed in this project.

*Results* — As a result of this project, we have a working prototype of a decentralized application that records cargo containers events and allows direct payments from container owners to service providers using blockchain technology.

*Further research* — The inclusion of IoT technologies would be an obvious further step on this research. There is also the possibility to add more layers of data to the smart contract to have a more detailed information about the cargo carried inside the container. Data related to type of cargo, temperature, customs, sanitary documentation, etc. can be also traced. Including other stakeholders such as customs and sanitary authorities to facilitate customs clearance. The creation of a custom currency to avoid variations on ether valuation can be also considered.

*Abstract* — This dissertation explores how blockchain technology can be used to improve assets tracking in the supply chain. Tracking cargo containers is still a challenge for the logistics industry. Container owners rely on direct links with each participant in the supply chain to track their containers. The information exchanged between container owners and service providers is later used as baseline for cargo tracking and invoicing. On numerous occasions, discrepancies between systems lead to lost containers and legal disputes. Instead of recording container status on different systems, this work proposes the use of blockchain technology to record container events into a single immutable ledger. After a theoretical introduction to the technologies that support blockchain platforms, it is described the architecture of a decentralized application based on Ethereum platform. Finally, a decentralized application prototype based on the previously proposed architecture has been implemented. The prototype demonstrated the validity of the principles collected throughout this work and can be a starting point for future developments.

Autor: Carlos Jácome Ferrao, email: carjafer@teleco.upv.es
Director: Manuel Esteve Domingo, email: mesteve@dcom.upv.es
Fecha de entrega: 25-11-2020

# INDEX

**I. Introduction** .......... **4**

**II. State of the Art** .......... **5**

II.1. Transactions .......... 5

II.2. Consensus Mechanism .......... 6

II.3. Merkle Tree .......... 6

II.4. Peer-to-Peer Network .......... 7

II.5. Scripting .......... 7

**III. Architecture** .......... **9**

III.1. Ethereum Accounts .......... 9

III.2. Messages and Transactions .......... 10

III.3. Ethereum Virtual Machine (EVM) .......... 10

III.4. Smart Contracts .......... 11

III.5. User Interface .......... 11

III.6. Data Storage .......... 12

**IV. Prototype** .......... **13**

IV.1. Business Scenario .......... 14

IV.2. Solution .......... 14

IV.3. Demo .......... 18

**V. Conclusions** .......... **25**

**References** .......... **26**

# I. Introduction

Nowadays, freight containers tracking represents a challenge for the logistics industry. Container owners rely on direct links with each participant in the supply chain to record changes in the status of cargo containers. These events are later used as baseline for cargo tracking and invoicing. On numerous occasions, discrepancies between systems lead to lost containers and legal disputes.

This thesis proposes the use of blockchain technology to track cargo containers along its journey from its inclusion into the equipment pool to its retirement at the end of its lifecycle. Instead of recording updates to container status on different systems, blockchain technology allows to record container events into a single immutable ledger, adding transparency and traceability to the entire system.

The proposed solution is a decentralized application on a permissioned blockchain, where only registered participants can interact with the application. The participants interact with the blockchain through a user interface that presents relevant information according to the participant profile (e.g. cargo owner, terminal, warehouse, customs). Each participant is responsible for recording container events that take place in their premises on the blockchain.

Additionally, the decentralized application allows automatic payments for the services incurred during the container journey. For each event recorded on the blockchain, a smart contract calculates the amount of money that must be payed to the service provider (e.g., terminal, depot or warehouse). Container owners will be able to validate the amount and pay inside the application just few minutes after the service has been completed. Usually, this payment process takes days or even weeks.

For this solution to work, the selected blockchain platform must give the possibility to develop business logic through smart contracts and provide a native cryptocurrency to implement automatic payments.

## II. State of the Art

A chain of transaction blocks linked by previous block hash was first described by Satoshi Nakamoto on his paper published on 2008[1], introducing a new digital currency called Bitcoin.

Nakamoto's purpose with Bitcoin was to enable an innovative peer to peer platform to transfer electronic currency without a central authority, by implementing software programs for validation, verification, and consensus on a decentralized infrastructure. Computation elements were added later to the blockchain infrastructure that has opened new possibilities beyond currency transfer.

Bitcoin supports an optional and special feature called *scripts* for conditional transfer of values. Ethereum, a second generation blockchain, extended this scripting feature into a complete code execution framework called smart contracts. Ethereum provides a blockchain with a Turing-complete programming language that can be used to create smart contracts.

### II.1. *Transactions*

From a technical point of view, we can think of Bitcoin as a state transition system, where we have a state consisting of the ownership of all existing bitcoins and, when applying a transition function that takes current state and a number of transactions, outputs a new state as the result. Take for example a standard banking system, the state is a balance sheet, the transaction is a request to move certain amount of money from one account to another, the transition function decreases the balance in the sender's account, and increases the balance in the receiver's account by the amount requested in the transaction. If the sender's account does not have enough funds requested in the transaction, the transition function returns an error [2].

In Bitcoin, the state is the collection of all unspent transaction outputs (UTXO), with each UTXO having a denomination and an owner. The owner is represented by a 20-byte address, which is essentially a cryptographic public key. Each transaction contains one or more inputs, a cryptographic signature produced with the owner's address private key, and one or more outputs. In the same way that each transaction input is associated to an existing UTXO, each output is an UTXO added to the new state [2].

Transactions are grouped into blocks, each block containing a reference to previous block hash, a timestamp, a nonce, and a list of all transactions that have taken place since the previous block. Over time, this creates a persistent blockchain that continually updates to maintain the digital ledger up to date [2].

## II.2. *Consensus Mechanism*

For all nodes to agree on the order of the blocks in the blockchain, Bitcoin needed to combine the state transition system with a consensus mechanism. The proof-of-work is the consensus mechanism used by Bitcoin and Ethereum.

Proof-of-work implementation involves scanning for a value that when hashed, it begins with a number of zero bits. The block's nonce is incremented until a value is found that produces a hash with the expected number of zero bits. The average work required is exponential in the number of zero bits required and can be verified by executing a single hash. This makes proof-of-work hard to calculate, but easy to verify. Once the block is generated, it cannot be changed without doing the proof-of-work again [1].

Determining the representation in majority decision making is also solved through the proof-of-work. The majority is represented by the longest chain, which is the greatest proof-of-work effort invested in it. If the majority of CPU is controlled by honest nodes, any competing chains will be outpaced by the honest and fastest growing chain. Attackers wanting to modify a past block will have not only to redo the proof-of-work of the targeted block, but also the proof-of-work of all blocks after it, and to surpass the work of honest nodes [1].

The process of creating new blocks is called mining, therefore, block creators are miners. Miners are entitled to 12.5 BTC for every valid block added to the chain as a compensation fee for the computational work expended. Additionally, in the case the value of the inputs in a transaction is higher than the value of the outputs, the miner gets the difference as a transaction fee [2].

## II.3. *Merkle Tree*

In bitcoin, the hash of a block, a 200-byte piece of data, is in fact, the hash of the block header. It contains the previous block hash, the timestamp, the nonce and the root hash of a data structure called the Merkle tree representing all the transactions in the block [2].

A Merkle tree is a type of binary tree, where each transaction is hashed, then two transaction hashes are concatenated and hashed together, and so on until there is a single hash that represents all the transactions in the block. The reason why this works is that hashes propagate upward: if a malicious user attempts to change a transaction in the bottom of the Merkle tree, this change will propagate to the nodes above, all the way to the root hash and therefore the hash of the block, causing the protocol to register it as a completely different block [2].

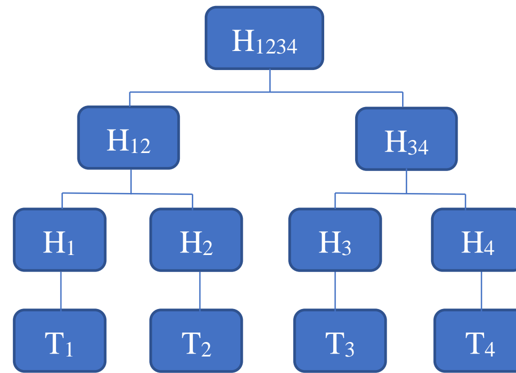


Fig.1. Merkle Tree.

The purpose of the Merkle tree is to allow the data in a block to be delivered in small pieces, a node can download only the header of a block from one source, the small part of the tree relevant to them from another source, and still be assured that all of the data is correct. The Merkle tree protocol is an important scalability feature of Bitcoin, essential to long-term sustainability [2].

### II.4. *Peer-to-Peer Network*

New transactions are sent to all nodes, each node adds the new transactions to a block and starts the work to find the proof-of-work. A new transaction does not need to reach all nodes, as long as they reach many nodes, it will be added to a block sooner than later [1].

New created blocks are broadcasted to all nodes once the proof-of-work is found, the new block is accepted only if all transactions are valid. Nodes do not have to communicate that the new block has been accepted, they just start working on the next block using the hash of the previous block. Blocks are also tolerant to dropped messages, nodes will realize that one block is missing once the next one is received, in that case the node requests the missing block to complete the chain [1].

In the case two nodes broadcast different versions of the same block at the same time, all nodes will start working on the branch formed by the block that was received first, and save the other block in case the branch becomes longer. The longest chain is always considered to be the correct one. When the next proof-of-work is found and one of the branches becomes longer, the nodes working in the other branch will switch and start working on the longer branch [1].

### I.5. *Scripting*

The concept of smart contracts was first proposed by Nick Szabo in 1994 [3]. The idea behind the smart contract is self-executing contracts, written in the form of software programs, with the terms of the agreement between interested parties. A smart contract is a piece of software that exist across

a distributed, decentralized blockchain network and allows transactions to be conducted between anonymous or untrusted parties without the need for a central authority [5].

Bitcoin protocol provides a basic version of the concept of smart contracts, where a UTXO can be owned not only by a public key, but also by a more complicated script expressed in a simple stack-based programming language. In this model, a transaction must provide the data to satisfy the script in order to spend the UTXO. Another example is a script that requires more than one private key signature to validate the transaction [2].

However, writing smart contracts with complex logic is not possible in Bitcoin due to the limitations of its scripting language [2]. Second generation blockchain platforms, such as Ethereum, embrace the idea of running user-defined software programs on the blockchain, thus creating expressive customized smart contracts with the help of Turing-complete programming languages [5]. Ethereum smart contracts are written in high level languages, such as Solidity, and compiled in a stack-based bytecode language and executed in Ethereum Virtual Machine (EVM). Ethereum is currently the most popular platform for developing smart contracts.

## III. Architecture

A decentralized application solves a problem that requires blockchain services and blockchain infrastructure for realizing its purpose. Typically, a decentralized application has a web frontend, a blockchain backend, and the code connecting the two.

In such an architecture, the frontend of a decentralized application channels any external stimulus from the users to the blockchain infrastructure and returns any response back to them.

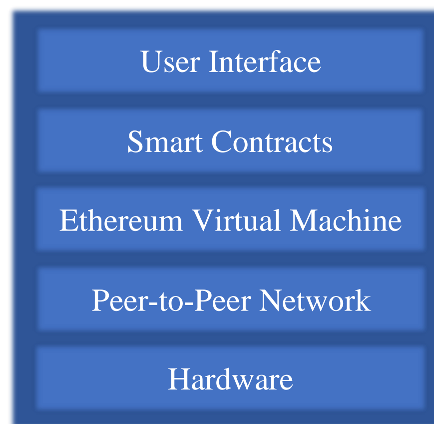


Fig.2. Ethereum architecture stack.

### III.1. *Ethereum Accounts*

In Ethereum, the state is made up of objects called accounts, with each account having a 20-byte address, and state transitions being direct transfers of value and information between accounts [2].

An Ethereum account contains four fields: the nonce, the account's ether balance, the contract code (if present) and the account's storage (empty by default). The nonce is a counter used to ensure each transaction is only processed once. Ether is Ethereum's built-in cryptocurrency, and it is used to pay transaction fees [2].

In general, there are two types of accounts: externally owned accounts or EOA, controlled by private keys, and contract accounts, controlled by their contract code. An externally owned account has no code, and one can send messages from an externally owned account by creating and signing a transaction; in a contract account, every time the contract account receives a message its code activates, allowing it to read and write to internal storage and send messages to other contracts [2].

### *III.2. Messages and Transactions*

The term transaction is used in Ethereum to refer to the signed data package that stores a message to be sent from an externally owned account. Each transaction contains, the recipient of the message, the signature that identifies the sender and the amount of ether to transfer from the sender to the recipient. Transactions also contain an optional data field, STARTGAS and GASPRICE values [2].

The first three fields are expected in any cryptocurrency system. The data field has no function by default, but the virtual machine has an opcode which a contract can use to access the data. The STARTGAS value represents the maximum number of computational steps that the transaction execution can take. The GASPRICE represents the fee the sender pays per computational step [2].

In order to prevent intended or accidental infinite loops or computational expensive operations that may slow down the network, each transaction must set a limit of computational steps of code execution that can be used. Computational cost is calculated in gas units, usually one computational step costs one gas. This means that computationally more expensive operations cost higher amounts of gas that simple ones. There is also a fee of five gas for every byte added to the transaction data [2].

### III.3. *Ethereum Virtual Machine (EVM)*

At the heart of the Ethereum protocol and operation is the Ethereum Virtual Machine, or EVM. The EVM is a computation engine, not quite different to the virtual machines of Microsoft's .NET Framework, or interpreters of other bytecode-compiled programming languages such as Java [6].

The EVM is the part of Ethereum that handles smart contract deployment and execution. At a high level, the EVM running on the Ethereum blockchain can be thought of as a global decentralized computer containing millions of executable objects, each with its own permanent data store [6]. The code in Ethereum contracts is converted into a low-level bytecode language. This stack-based code is also known as Ethereum virtual machine code or EVM code. The EVM code consists of a series of bytes, where each byte represents an operation. EVM code execution is an infinite loop that consists of performing the operation pointed to by the program counter and then incrementing the program counter by one, until the end of the code is reached, or an error occurs. [2].

The operations have access to three types of space to store data: stack, memory and storage. The stack is a last-in-first-out container limited in size, to which values can be pushed and popped. Memory space is an infinitely expandable not persistent byte array, when the contract execution finishes memory contents are not saved. Unlike stack and memory, which reset after computation ends, storage persists for the long term. Contract storage is a key/value store that acts as a public

database, from which values can be read externally without having to send a transaction to the contract. However, writing to contract storage is expensive compared to writing to memory. The code can also access the value, sender and data of the incoming message, as well as block header data, and can return a byte array of data as an output [2].

### III.4. *Smart Contracts*

In decentralized applications, smart contracts are used to store the business logic and the related state of the application. Smart contracts can be viewed as the server-side (backend) component in a regular application. One of the main differences is that any computation executed in a smart contract is expensive and so should be kept as minimal as possible. It is therefore important to identify which aspects of the application need a trusted and decentralized execution platform [6].

Ethereum blockchain platform allows to build architectures in which a network of smart contracts call and pass data between each other, reading and writing their own state variables, with their complexity restricted only by the block gas limit. One major consideration of smart contract architecture design is that it is not possible to change the smart contract code once it is deployed. It can be deleted if it is programmed with an accessible SELFDESTRUCT opcode, but it cannot be changed in any way [6].

The second major consideration of smart contract architecture design is the application size. A large monolithic smart contract may cost a lot of gas to deploy and use. Therefore, some applications may choose to have off-chain computation and an external data source [6].

### III.5. *User Interface*

Unlike the business logic of a decentralized application, which requires to understand the EVM and new programming languages such as Solidity, the client-side interface uses standard web technologies HTML, CSS and JavaScript. Interactions with Ethereum, such as signing messages, sending transactions, and managing keys, are often conducted through the web browser, via an extension such as MetaMask [6].

Although it is possible to create a mobile decentralized application, currently there are few resources to help create mobile decentralized applications frontends, mainly due to the lack of mobile clients that can serve as a light client with key-management functionality. The frontend is usually linked to Ethereum via the web3.js JavaScript library, which is bundled with the frontend resources and served to a browser by a web server [6].

Web3.js is a collection of libraries that allow you to interact with a local or remote Ethereum node using HTTP, IPC or WebSocket. It represents a JavaScript language binding for Ethereum JSON

RPC interface, which makes it directly usable in web technology, as JavaScript is natively supported in almost all web browsers. Using MetaMask browser extension in combination with web3.js, in a web interface, is a convenient way to interact with the Ethereum network.

### III.6. *Data Storage*

Due to high gas costs and the currently low block gas limit, smart contracts are not well suited to storing or processing large amounts of data. Hence, most decentralized applications utilize off-chain data storage services, storing the bulky data outside the Ethereum blockchain, on a data storage platform. The data storage platform can be centralized (cloud database), or decentralized, stored on a P2P platform such as the IPFS or Swarm [6].

Decentralized P2P storage is ideal for storing and distributing large static assets such as images, videos, and the resources of the application's frontend web interface (HTML, CSS, JavaScript).

## IV. Prototype

In order to demonstrate the validity of the principles collected in previous sections, a decentralized application prototype has been developed. This prototype is intended as a proof of concept and starting point for future developments.

A representative number of events were selected to track containers status and pay for the services received. These events will be represented as functions in the smart contract. A web page is used as the user interface that allows the container owner and service providers to interact with the smart contract.

The user interface provides information about the user account, account balance, container numbers, container status and location. In addition, the container owner can see the amount of ether owed to the providers, pay for the services received, and display the events recorded in the blockchain.

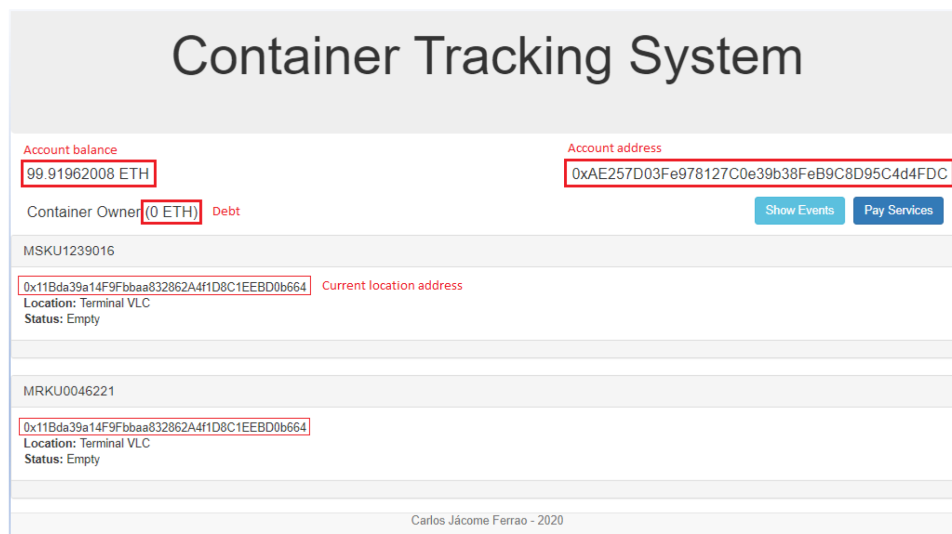


Fig.3. Decentralized Application User Interface.

Service providers can see the containers located on their premises or in transit, and the amount of ether owed to them. They can also record container events through a set of buttons showed in the user interface at container level.

Truffle framework has been used to develop the decentralized application prototype. Truffle provides a development tool with the ability to test and implement decentralized applications based on Ethereum blockchain. Ganache, also part of Truffle framework, is a local blockchain simulator that allows to replicate blockchain networks to test smart contracts.

## III.1. *Business Scenario*

This section describes the business scenario that will be used later to run a demo of the prototype.

Consider two containers, MSKU1239016 and MRKU0046221, that will be exported from Valencia, Spain to Cartagena, Colombia. Both containers will be initially located at a container terminal in Valencia (Terminal VLC). From there, the containers will be picked up by a truck and delivered to a warehouse within Valencia city (Warehouse VLC). Inside the warehouse, the containers will be stuffed with the cargo that will be later imported to Colombia.

The two full containers will be transported back to the terminal by truck to be loaded on a vessel that will take them to Cartagena. Once discharged in Cartagena (Terminal CTG), the containers will be picked up by a truck and moved to a warehouse for delivering the cargo (Warehouse CTG).

Both empty containers are transported back to the container terminal for storage, waiting for the next shipment. Finally, the container owner will pay for the fees incurred during the container transportation to the service providers, i.e. terminals and warehouses.

## III.2. *Solution*

The smart contract is the piece of software that handles decentralized application's backend business rules. On our prototype, the *ContainerTracker* smart contract is written in Solidity, an object-oriented, high-level language created for implementing smart contracts on Ethereum blockchain.

In the first section of the smart contract, custom variables that represent the containers and service providers locations were defined as shown in Fig. 4. *Container* structure includes variables that identify current location, previous location, full or empty status, and transport mode. Transport mode can be "truck" or "vessel" when the container is being moved from one location to another, and "none" when the container is inside a terminal or warehouse.

*Location* structure includes variables to identify the location name, location type and country. Status, location mode, transport mode and country are lists of fixed values that identify attributes of locations and containers. Container owner, debt to vendors, and lists of containers and locations are also defined as variables.

In Solidity language, modifiers are inheritable properties that allow to control the behaviour of the functions in the smart contract. Modifier *onlyOwner*, described in Fig. 5, requires that only the container owner account calls a specific function. The other two modifiers described in Fig. 5 validate that containers and locations are not registered more than once.

```
// custom types
struct Container {
    address location;
    address prevLoc;
    Status status;
    TransportMode transportMode;
}

struct Location {
    string locName;
    LocationType locType;
    Country country;
}

enum Status {empty, full}
enum TransportMode {none, truck, vessel}
enum LocationType {undef, terminal, warehouse}
enum Country {undef, es, co, mx}

// state variables
address payable owner;
string[] public containerIds;
mapping (string => Container) public containers;
address[] public locationIds;
mapping (address => Location) public locations;
mapping (address => uint256) public vendorDebt;
```

Fig.4. Smart contract custom types and state variables.

```
// modifiers
modifier onlyOwner() {
    require(msg.sender == owner, "This function can only be called by the contract owner");
    _;
}

modifier checkContainerExists(string memory _containerId) {
    bool isAdded = false;
    for(uint i = 0; i < containerIds.length; i++) {
        if(keccak256(bytes(containerIds[i])) == keccak256(bytes(_containerId))) {
            isAdded = true;
            break;
        } else continue;
    }
    require(isAdded == false, "Container is already in the pool.");
    _;
}

modifier checkLocationExists(address _location) {
    bool isAdded = false;
    for(uint i = 0; i < locationIds.length; i++) {
        if(locationIds[i] == _location) {
            isAdded = true;
            break;
        } else continue;
    }
    require(isAdded == false, "Location is already registered.");
    _;
}
```

Fig.5. Smart contract modifiers.

Functions *addLocation* and *addContainer* are used to register new locations and add new containers to the pool. Both functions verify that new values are not registered already by inheriting the modifiers that check for duplicates. Functions *addLocation* and *addContainer* can be called only by the container owner.

```
// add locations to the address book
function addLocation(address _location, string memory _locName, LocationType _locType, Country _country)
public checkLocationExists(_location) onlyOwner() {
    locationIds.push(_location);
    locations[_location] = Location(
        _locName,
        _locType,
        _country
    );
}

// add containers to the pool
function addContainer(string memory _containerId, address _location)
public checkContainerExists(_containerId) onlyOwner() {
    containerIds.push(_containerId);
    containers[_containerId] = Container(
        _location,
        address(0),
        Status.empty,
        TransportMode.none
    );
}
```

Fig.6. Smart contract functions *addLocation* and *addContainer*.

Functions *gateIn* and *gateOut* record transactions when the containers arrive or leave a location through the gate. Function *gateOut* first validates that the container is currently in the location that called the function. Then, current location is updated to zero, meaning that the container is in transit, and previous location to the address that called the function; transport mode is updated with the value "truck". Note that one ether is added to the *vendorDebt* variable.

Function *gateIn* validates that the container is being carried on a truck and that the previous location is not the same that calls the function. If both validations are false, current location is updated with the address that called the function, transport mode is set to none and location vendor debt is increased by one ether.

```
// container gate out
function gateOut(string memory _containerId) public {
    require(containers[_containerId].location == msg.sender, "Container is not in this location.");

    containers[_containerId].location = address(0);
    containers[_containerId].prevLoc = msg.sender;
    containers[_containerId].transportMode = TransportMode.truck;
    vendorDebt[msg.sender] += 1 ether;

    emit LogContainerEvent(_containerId, msg.sender, "Gate-Out");
}

// container gate in
function gateIn(string memory _containerId) public {
    require(containers[_containerId].transportMode == TransportMode.truck, "Container is not in transit.");
    require(containers[_containerId].prevLoc != msg.sender, "Container cannot gate in on the same location.");
    containers[_containerId].location = msg.sender;
    containers[_containerId].transportMode = TransportMode.none;
    vendorDebt[msg.sender] += 1 ether;

    emit LogContainerEvent(_containerId, msg.sender, "Gate-In");
}
```

Fig.7. Smart contract functions *gateOut* and *gateIn*.

Functions *stuffing* and *stripping* change the container status, full or empty, accordingly. Both functions validate that the container is in the location that calls the function and current container status, the container must be full to be stripped and vice versa. Both events have a cost of one ether to the container owner.

```
// container stuffing
function stuffing (string memory _containerId) public {
    require(containers[_containerId].location == msg.sender, "Container is not in this location.");
    require(containers[_containerId].status == Status.empty, "Container should be empty.");
    containers[_containerId].status = Status.full;
    vendorDebt[msg.sender] += 1 ether;

    emit LogContainerEvent(_containerId, msg.sender, "Stuffing");
}

// container stripping
function stripping (string memory _containerId) public {
    require(containers[_containerId].location == msg.sender, "Container is not in this location.");
    require(containers[_containerId].status == Status.full, "Container should be full.");
    containers[_containerId].status = Status.empty;
    vendorDebt[msg.sender] += 1 ether;

    emit LogContainerEvent(_containerId, msg.sender, "Stripping");
}
```

Fig. 8. Smart contract functions *stuffing* and *stripping*.

*Discharge* function records container arrivals to terminals through a vessel. It validates that the container transport mode is “vessel” and that the previous location is different from the one calling the function. Similarly, *load* function records events of containers leaving the terminal on a vessel. It validates that the container is in the same location that called the function. Both functions have a of cost two ether to the container owner.

```
// container load
function load(string memory _containerId) public {
    require(containers[_containerId].location == msg.sender, "Container is not in this location.");
    containers[_containerId].location = address(0);
    containers[_containerId].prevLoc = msg.sender;
    containers[_containerId].transportMode = TransportMode.vessel;
    vendorDebt[msg.sender] += 2 ether;

    emit LogContainerEvent(_containerId, msg.sender, "Load");
}

// container discharge
function discharge(string memory _containerId) public {
    require(containers[_containerId].transportMode == TransportMode.vessel, "Container must be on a vessel.");
    require(containers[_containerId].prevLoc != msg.sender, "Container cannot be discharged on the same location it was loaded.");
    containers[_containerId].location = msg.sender;
    containers[_containerId].transportMode = TransportMode.none;
    vendorDebt[msg.sender] += 2 ether;

    emit LogContainerEvent(_containerId, msg.sender, "Discharge");
}
```

Fig.9. Smart contract functions *load* and *discharge*.

Finally, the function *initializeDemo* has been created to register four locations and add two containers to the pool. Ganache generates ten testing accounts with 100 ether each. First account is reserved for the container owner. The next four accounts are registered as the locations that will handle the containers.

```
// initialize contract for demo
function initializeDemo() public {
    addLocation(0x11Bda39a14F9Fbbaa832862A4f1D8C1EEBD0b664, "Terminal VLC", LocationType.terminal, Country.es);
    addLocation(0xBD2b9c7fE5B85f3874f33cDcFa970E64554Ad5B4, "Warehouse VLC", LocationType.warehouse, Country.es);
    addLocation(0xdDA691FD4664b0Eca4FFDeaE529996905C3EA789, "Terminal CTG", LocationType.terminal, Country.co);
    addLocation(0xf260aa16Ca032acd59F228ace12974Bd8872CFd9, "Warehouse CTG", LocationType.warehouse, Country.co);
    addContainer("MSKU1239016", 0x11Bda39a14F9Fbbaa832862A4f1D8C1EEBD0b664);
    addContainer("MRKU0046221", 0x11Bda39a14F9Fbbaa832862A4f1D8C1EEBD0b664);
}
```

Fig.10. Smart contract, function *initializeDemo*.

In Fig.11, Ganache user interface shows the five accounts that have been registered to run the demo. Container owner's account has been used to deploy the smart contract, note that its balance has been decreased in few decimals.

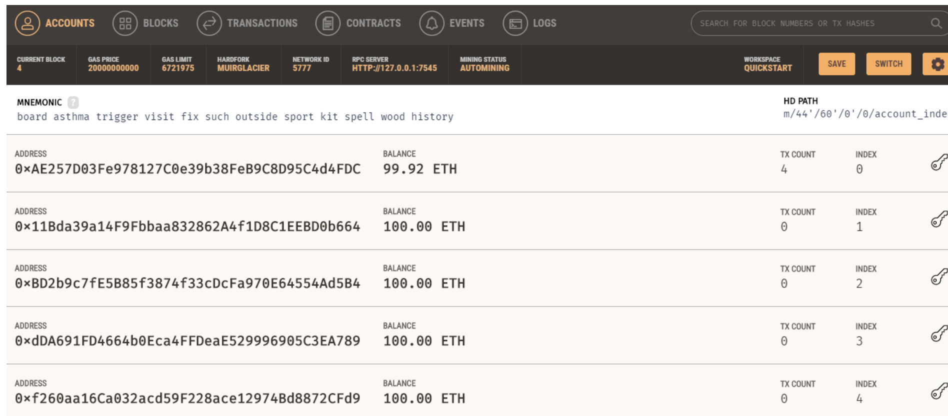


Fig.11. Ganache, locations accounts.

## III.3. *Demo*

In this section, a demo of the prototype will be run following the business scenario described in previous sections. Initially, both empty containers are in Terminal VLC. The terminal can take two actions, either load the empty containers onto a vessel or gate them out on a truck. The containers are visible only to the container owner and Terminal VLC.

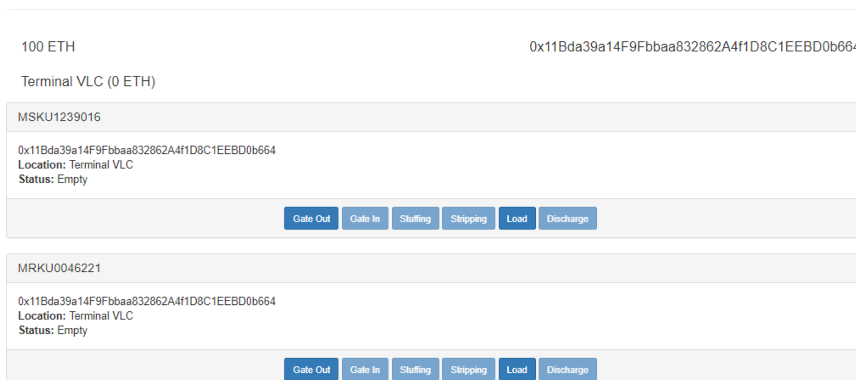


Fig.11. Terminal VLC, empty containers.

In order to follow our business scenario, the containers will be taken to a warehouse for stuffing. All transactions must be confirmed in MetaMask to be recorded in the blockchain. As shown in Fig. 12, there is no currency transfer in this transaction, but only changes on containers variables.

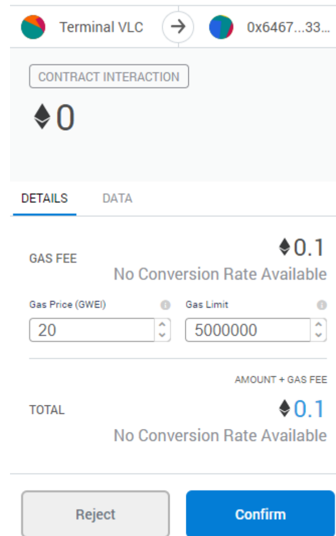


Fig.13. MetaMask, transaction confirmation.

Once the transactions are confirmed, gate out events are recorded in the blockchain and the containers are no longer in Terminal VLC, but in transit on a truck. At this point, the containers are visible to the container owner and the service providers located close to the terminal. Notice that the amount of ether owed to Terminal VLC has been increased by two, one ether per gate out event.

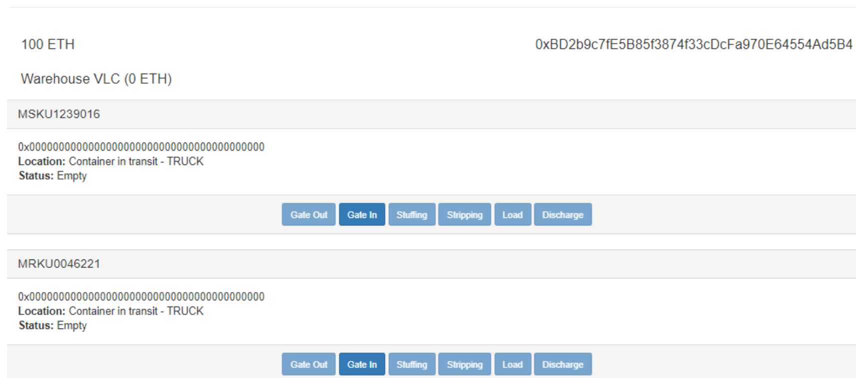


Fig.14. Warehouse VLC, empty containers in transit (truck).

Ganache user interface allows to review the block mined and the transaction included in the block as shown in Fig. 4.13. In Ganache, the process of mining a new block takes few seconds, each block usually includes only one transaction.

Fig.15. Ganache. Block 5.

When the two containers gate into Warehouse VLC, the amount of debt to the location increases in two ether. Containers are now visible to Warehouse VLC and the container owner.

Fig.16. Warehouse VLC, empty containers gated in.

The containers are stuffed inside Warehouse VLC. Each stuffing event increases the debt count in one ether. Once full, the containers gate out from Warehouse VLC on a truck and gate in Terminal VLC to be loaded onto a vessel. Each gate event increases the debt by one ether.

Fig.17. Terminal VLC, full containers in transit (truck).

99.9952726 ETH 0x11Bda39a14F9Fbbaa832862A4f1D8C1EEBD0b664

Terminal VLC (4 ETH)

MRKU0046221

0x11Bda39a14F9Fbbaa832862A4f1D8C1EEBD0b664
**Location:** Terminal VLC
**Status:** Full

Gate Out Gate In Stuffing Stripping Load Discharge

MSKU1239016

0x11Bda39a14F9Fbbaa832862A4f1D8C1EEBD0b664
**Location:** Terminal VLC
**Status:** Full

Gate Out Gate In Stuffing Stripping Load Discharge

Fig.18. Terminal VLC, full containers gated in.

Containers are loaded on a vessel in Terminal VLC. Each load move has a cost of two ether to the container owner. The containers are now visible to the container owner and Terminal CTG.

100 ETH 0xdDA691FD4664b0Eca4FFDeaE529996905C3EA789

Terminal CTG (0 ETH)

MSKU1239016

0x0000000000000000000000000000000000000000
**Location:** Container in transit - VESSEL
**Status:** Full

Gate Out Gate In Stuffing Stripping Load Discharge

MRKU0046221

0x0000000000000000000000000000000000000000
**Location:** Container in transit - VESSEL
**Status:** Full

Gate Out Gate In Stuffing Stripping Load Discharge

Fig.19. Terminal CTG, containers in transit (vessel).

The containers are discharged in Terminal CTG, each discharge move has a cost of two ether to the container owner. On this stage, the containers are only visible to the container owner and the Terminal CTG.

99.99719864 ETH 0xdDA691FD4664b0Eca4FFDeaE529996905C3EA789

Terminal CTG (4 ETH)

MRKU0046221

0xdDA691FD4664b0Eca4FFDeaE529996905C3EA789
**Location:** Terminal CTG
**Status:** Full

Gate Out Gate In Stuffing Stripping Load Discharge

MSKU1239016

0xdDA691FD4664b0Eca4FFDeaE529996905C3EA789
**Location:** Terminal CTG
**Status:** Full

Gate Out Gate In Stuffing Stripping Load Discharge

Fig.20. Terminal CTG, full containers discharged.

Later, both containers are taken to Warehouse CTG for stripping. Each gate transaction increases the amount of debt by one ether. Containers go first on an intermediate state, in transit, where they are only visible to the owner and the facilities near the terminal.

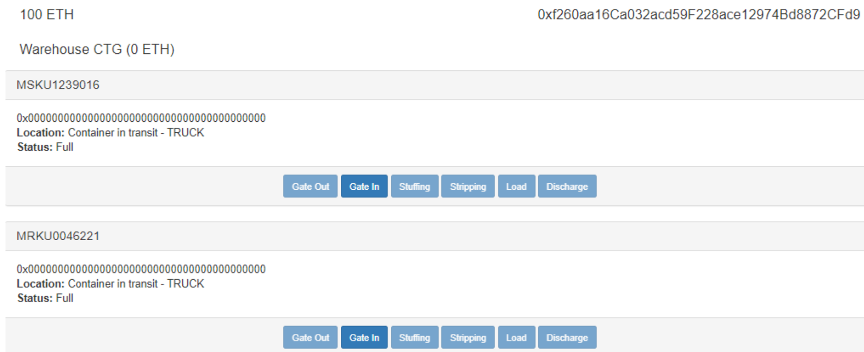

Fig.21. Warehouse CTG, full containers in transit (truck).

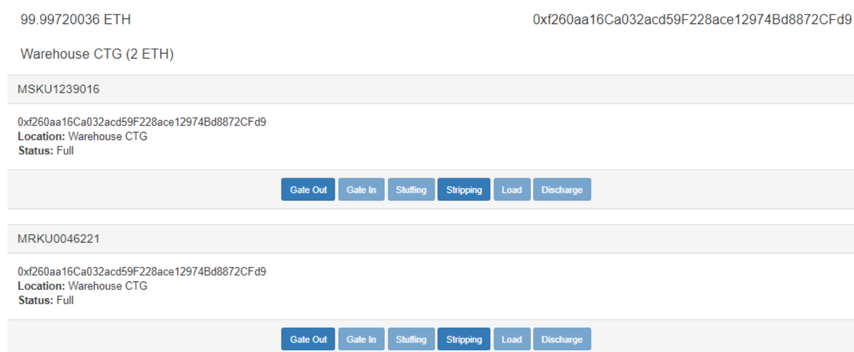

Fig.22. Warehouse CTG, full containers gated in.

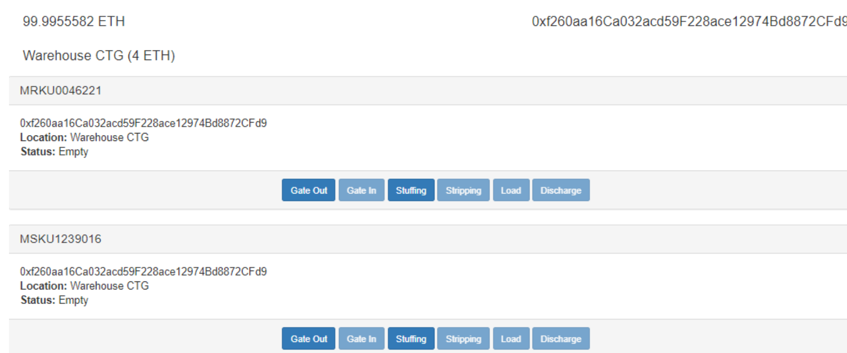

Fig.23. Warehouse CTG, empty containers after stripping.

Once the containers are empty, they are sent back to Terminal CTG. In Terminal CTG, the containers are stored waiting for the next shipment. Each gate transaction increased the debt count in one ether. Final debt count is 28 ether, container owner owes eight ether to each terminal and 6 ether to each warehouse.

Fig.24. Terminal CTG, containers in transit.



Fig.25. Container Owner, empty containers located in Terminal CTG.

Each container move generates an event that is recorded in the blockchain. All container events are visible to the container owner from the decentralized application UI selecting the button *Show Events*. Reviewing the records, the container owner can track each container from its inclusion to container pool.



Fig.26. User Interface. Containers events (Show Events).

Container owner can use the *Pay Services* button to pay to the service providers for the costs incurred during the container journey. Each payment, as any other transaction, will require to be confirmed in MetaMask before executing. Container owner must pay eight ether to each terminal and 6 ether to each warehouse.

Fig.27. MetaMask, transaction confirmation (payment).

Block 29 shows one of the payment transactions, it transfers 8 ether to Terminal VLC for the services provided to containers MSKU1239016 and MRKU0046221.

Fig.28. Ganache. Block 29 containing payment transaction.

Fig.29. Ganache. Payment transaction.

## V. Conclusions

Throughout this work, a prototype of a decentralized application has been implemented to track cargo containers and pay for the services incurred during the containers journey. Instead of recording container status on different systems, this thesis proposes the use of blockchain technology to record container events into a single immutable ledger. Additionally, using Ethereum built-in cryptocurrency, ether, container owners can pay for the services provided inside of the same application, automatically and in a decentralized way. This means that the payments are done transferring the amount of ether from one Ethereum account to another without the intervention of any central entity. The architecture described in this document demonstrates that blockchain technology is a suitable solution to track cargo containers facilitating transparency and traceability.

The inclusion of IoT technologies appears to be an obvious further step on this research. Sensors can give more information in real time about the container status, being particularly relevant on reefer containers, used to transport perishables and dangerous goods. Following the same path, it is possible to add more layers of data to the smart contract to have more detailed information about the cargo carried by the container. Data related to type of cargo, temperature, customs, sanitary documentation, etc. can also be tracked along with the containers.

In other to broaden the scope of the solution, other stakeholders can be invited to participate such as customs and sanitary authorities to facilitate cargo customs clearance. Finally, the creation of a custom cryptocurrency to pay for the services provided to avoid variations on ether valuation can be also considered to continue this work.

**REFERENCES**


[1] S. Nakamoto. *Bitcoin: A Peer to Peer Electronic Cash System*. 2008.

[2] V. Buterin. *A Next Generation Smart Contract and Decentralized Application Platform.* 2013.

[3] N. Szabo. *Smart Contracts: Building Blocks for Digital Markets*. 1996.

[4] A. Narayanan. J. Clark. *Bitcoin's Academic Pedigree: The concept of cryptocurrencies is built from forgotten ideas in research literature.* Communications of the ACM, December 2017.

[5] Shuai Wang, Yong Yuan, Xiao Wang, Juanjuan Li1, Rui Qin, Fei-Yue Wang. *An Overview of Smart Contract: Architecture, Applications, and Future Trends*. IEEE Intelligent Vehicles Symposium, 2018.

[6] A. M. Antonopoulos. G. Wood, *Mastering Ethereum: Building Smart Contracts and DApps*. 2018.