Decentralized Container Transport Market: A Hyperledger Fabric based solution

Algirdas Jokubauskas - 4744233
Blazej Kula - 4746724
Florian Unseld - 4698762
Philippe Louchtch - 4697685
Yugdeep Bangar - 4733967

February 2, 2018

Contents

1 Introduction

With over 7.4 million Twenty Foot Equivalent (TEU) containers the port of Rotterdam is the biggest Port in Europe. The port authority wants to enable a digital marketplace to match Logistics Service Providers (LSPs) with shipping containers for inland container transport. The LSP community is wary of centralized, Uber-style platform since it reduces profit margins for them. Therefore, possibilities are being explored to develop a decentralized marketplace for containers based on blockchain technology. This project is aimed to develop a prototype for such a marketplace. In chapter 2 the problem is further described, and in chapter 3 our solution to it is explained.

2 Problem Description

The problem for this group project is the organisation of a container transport market for the port of Rotterdam, such that the port of Rotterdam does not have to invest or maintain the system. A distributed blockchain system is proposed in which the involved parties can invest into the platform. This decentralised marketplace has two main aspects.

The first one is the matchmaking between available containers and available truckers. This matchmaking is supposed to increase the efficiency and decrease the time the container is at the respective terminal.

An Uber-style approach to this matchmaking is desired. Truckers look for a job in this case the delivery of a container on the blockchain. They bid on this job and the winner gets to deliver the container. The winner is chosen by the person offering the job according to their regulations.

The second aspect is the exception handling during or before the delivery. An exception in this context is an event preventing the trucker from completing the job on the shippers' terms. An example would be a traffic jam causing delay. These exceptions need to be logged ideally on the blockchain, to justify penalties or aid in lawsuits between a trucker and a container owner.

For usability a rating system, rating the trucker based on reliability or other key attributes is required.

An optional goal is taking traffic or weather information into account and automatically update the contract or raise exceptions. For completion legal binding contracts, and real-money bidding and paying should be implemented.

3 Solution

3.1 Platform Choice

Throughout the years various blockchain platforms have emerged. Each having unique features that would aid or hindrance the development of the described requirements. First and foremost, requirement is the blockchain has to be able to record the state of the market and record the execution of jobs. This is to meet the requirements of traceability of containers and for third parties to interact with the system. Furthermore, it is crucial to enforce all business rules of stakeholders and ensure validity of the data put on the chain. For this, a smartcontract-like functionality is required. The programming model and language must be straight forward and expressive.

We decided on a permissioned network to have more control over it, both management and administrative wise as well as security considerations. This also gives us independence from any price fluctuations of the platform. Another requirement is the proper support of smart contracts. These are as described in section 3.2 essential to our design.

A variety of platforms were considered, in the end, we decided to use Hyperledger Fabric as platform for this first prototype. It is able to achieve a high performance; the developing community is big enough to ensure stability to some degree and a stable version was already released. Furthermore Hyperledger Fabric Composer looked very developer-friendly and a promising way to kickstart development of a proof-of-concept such as this project.

3.2 System Design

3.2.1 High Level Introduction into Hyplerledger Fabric and Composer

Fabric is a distributed system for permissioned, private blockchain. It is highly modular as e.g.: a consensus mechanism or ledger maintenance library. Members of the network need to enroll via Membership Service Provider that provides identity for them. Each node takes up one of three roles:

- Client - proposes transactions,
- Peer - executes transactions and validates them,
- Orderer (OSN) - establishes total order of transactions.

Fabric contains a ledger that is composed from two components: world state database (current state of the ledger) and transaction log (history of all transactions). World state is altered by transactions written in chaincode which are then stored in the log. Chaincode can implemented in several programming languages (as of writing this Go, Node and Java).

Hyperledger Composer is a toolset and framework for quick blockchain application development that supports Hyperledger Fabric. Composer allows for quick business network modelling, it introduces custom languages for its key concepts (defined below) and allows to write transactions in JavaScript, which wasn't supported in Fabric when we started. In the essence it provides a high-level abstraction of the Fabric. We decided it was a good fit for our project as its main business scenario is similar to the distributed marketplace, it has adequate supporting tools, such as Playground environment and it allows for easy prototyping.

3.2.2 Concepts and Model

Terminology

The following list comprises the Composer and network specific concepts and terminology used in the later paragraph to describe the model itself. Most important are the following.

- Asset: Identifiable object stored in a registry comparable to an entity. It has a unique Id and can be queried.
- Participant: Participants are a special type of asset. It is bound to an identity and can follow additional rules. Their main purpose is to store some properties and to couple agents to assets.
- Concept: Data object or value object, which cannot be stored on its own. It is part of an Asset.
- Access Control Language: The Access Control Language (ACL) basically a file containing the rules to access assets commands and interactions. It is expressed with JavaScript conditions.
- Registry: Collection of assets of given type, it might contain all the assets of that type in the system or it might just contain the subset of it.
- Queries: They are used for queueing and filtering the results and they are defined in .qry files.
- Model File: File which contains definition of assets, participants, concepts and transactions.
- Chaincode Logic: Transaction code that runs on a blockchain.

Model

All of the assets in our network are declared in the /fabric_network/rotterdam_logistics_blocklab/ /models/nl.tudelft.blockchain.logistics.cto. Below we present most important ones:

- Trucker: Represents a participant that is a trucker. It contains specific information about the trucker, such as their rating and things such as ADR training level and their truck capacity.
- ContainerGuy: Represents a participant that is a shipper. It contains specific information about a shipper.
- ContainerInfo: Assets that represents a physical container and its attributes.
- ContainerDeliveryJobOffer: Asset that represents a container delivery job offer. It contains specific information about the delivery, such as: bids, destination, etc.
- ContainerDeliveryJob: Asset that represents a container delivery job in progress, it is created from a job offer after a bid has been accepted by the ContainerGuy (shipper). A Delivery Job is in fact a *contract* between the Shipper and Trucker. It can either be Delivered successfully or its delivery aborted through the raising of an *Exception* by either the Shipper or the Trucker.

Transactions

Transactions are declared in the same file as assets. They are annotated with `@transaction` and they defined are located in `/fabric_network/rotterdam_logistics_blocklab/lib/logic.js` directory. Please take a look at 1 for an impression of some of the main transactions and their effects as well as the transaction function code itself in the file described above. The transaction code is fairly clear and should be easily understood.

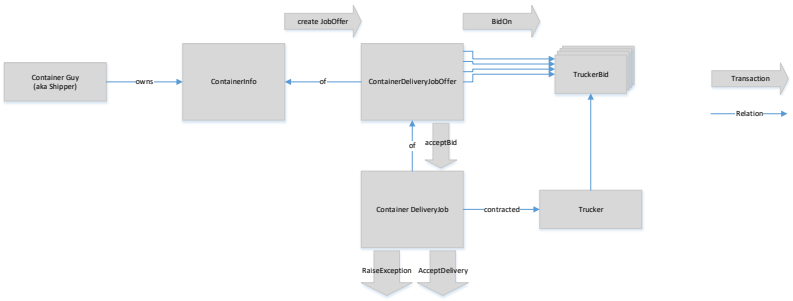

Figure 1: Assets life cycle and model

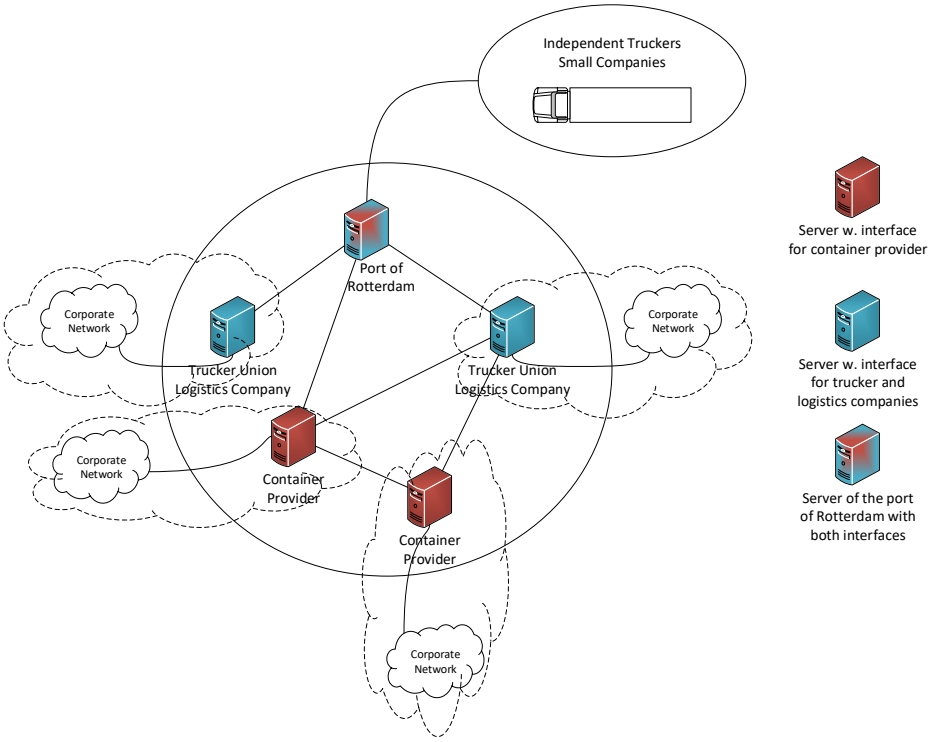

Figure 2: System Architecture

3.3 System Architecture

Due to how Hyperledger Fabric works, we envisioned that larger stakeholders in this system would eventually have their own nodes participating in the network, be it just peers or even orderers. Where even a distinction between Trucker and Shipper-focused nodes could be made, where each participating stakeholder could even be providing attestations for Shippers and Truckers through their nodes.

Here we also thought of possible node maintainers to increase the trustworthiness of the network, this is also illustrated in 2. But always there should be one neutral server and implementation run by the port of Rotterdam so small companies or freelancing truckers can access the system "vanilla".

Please note that the nodes are not ran by the participants on their devices, it is a server hosting some, or all Hyperledger Fabric processes.

The list of basic interactions of the two parties, the truckers and container providers is short. Those are shown in diagram 3. The figure also includes the interactions with the third party, the

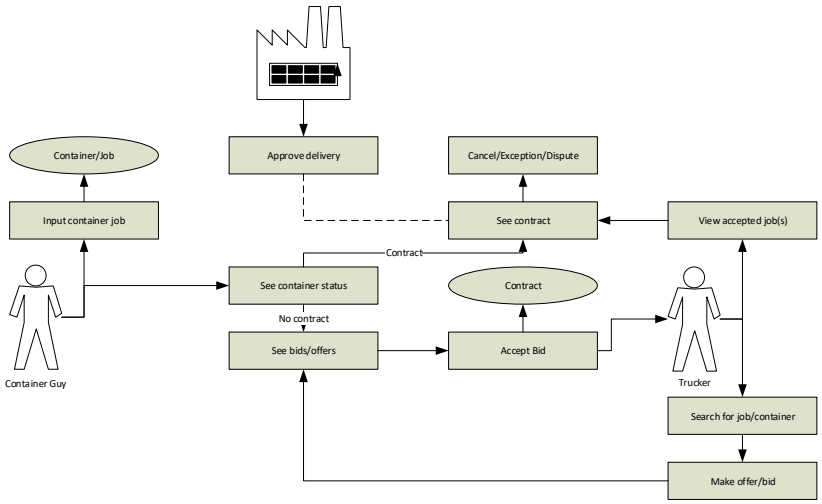

Figure 3: Interactions

recipient of the container.

It is important to note that during the development we were forced to migrate to Composer 0.17 and Fabric 1.1 as old versions introduced problems with queries and transactions.

We split the the system into three logical parts, the blockchain (with the chaincode), the RESTful middle-layer and the UI as outlined in 4.

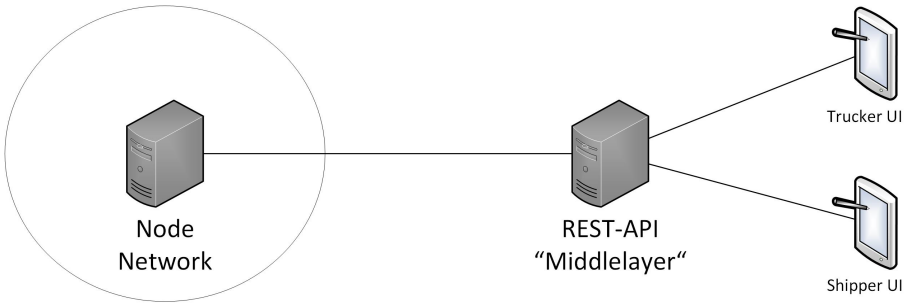

Figure 4: Current setup

3.3.1 Chaincode

Main goal of the chaincode is to modify the world state. It ensures that the data added to the blockchain is valid with respect to the business rules. It is written in Node.js. Due to the runtime cost of the chaincode, future iterations of this system are advised to do most of the validation and computation outside the chaincode, in the middleware such as the RESTful application in our case.

3.3.2 Backend

Our backend solution is a REST api developed in node.js. Different URLs are routed to separate classes based on logical models. These classes contain only endpoints and the logic is moved out into many service classes. All communication with the blockchain is handled by generic methods inside LogisticNetwork.js. For further information checkout the readme file in the repository.

3.3.3 Frontend

The frontend of our project is a ready to use user interface that can be used by truckers and shippers to connect to the REST server of Hyperledger fabric. It was developed using shiny framework (a high level abstraction of JavaScript functionalities) in R and has been hosted on RStudio webservers. It can be accessed on following URLs:

Shipper UI: https://yugdeep.shinyapps.io/ShipperUI/

Trucker UI: https://yugdeep.shinyapps.io/TruckerUI/

The frontend is basically a wrapper for data fetched from the blockchain using REST API calls. This is done in real-time and no data is stored on UI server. The shipper/trucker needs to provide their ID and the URL of Rest Server to interact with the blockchain.

The user interface is built using bootstrap framework which makes it compatible with mobile devices. Please do note that the front-end is not meant to be anything else than an UI for the proof-of-concept.

4 Features

One feature allows the trucker to enter destination (could be city name) and/or radius. This is then used to return containers that have their destinations in this region.

Automated rating is implemented. The blockchain tracks the amount of jobs that the trucker has succesffuly delivered and total amount of job that they have accepted in the past. This is used as a ratio that can guide the choice of the shipping company when browsing bids. We do not feel a star rating or a review system makes sense here. A logistics job is either executed well or not.

The chaincode handles business logic validation in the application. For example, after a shipping company accepts a bid, all other conflicting bids are automatically cancelled. It also provides detailed errors about what the user entered incorrectly which are then marshalled from the chaincode through the RESTful application to the UI.

5 Future work

In this chapter we will describe the steps that need to be taken to improve the platform and reach a usable state.

The groundwork for user access control is implemented, however due to lack of proper documentation we did not manage to connect to the network with a non-administrator account. Once this functionality is working properly, security and information hiding can be implemented through Composer's expressive *Access Control Language* although some model changes might be necessary.

Our current delivery job exception raising involves the trucker simply calling an exception and adding text info. It would be better to extend this functionality and add features like conflict resolution, third party involvement and others directly on the blockchain.

User interface is currently functional however it needs work in several areas. First of all, usability should be improved. This should be designed in cooperation with experts in the domain so that it would fit current workflow of shipping companies. Furthermore, a framework for error messages and texts around the application to be easily changed and translated into many languages.

There is now sufficienct groundwork laid to experiment with the marketplace. For example, one idea is to introduce a currency on the network, backed by euros 1:1 by Port of Rotterdam, and let any party bid on jobs. Turning the marketplace into a job exchange. The actors will be forced to play fair by maybe forcing them to stake some of their crypto money in every job they accept. These actors can then distribute these jobs between real Truckers he knows or works with. We should look for and experiment with solutions that are truly distruptive.

Just for any design, data security is a key feature in this design. However, time did not permit to address this problem properly. We were aware that the data of the trucker, of the container and of the shipper need to be partially visible yet secret, such that only relevant information is revealed, nothing more, even during attacks.

Depending on where the port wants to advance and experiment with this marketplace many new features and fundamental changes could be made, like trucker job requests. Here a trucker can

post his job preferences in advance and either be notified of jobs satisfying the search parameters or even automatically placing a bid if some conditions are met, further streamlining the marketplace.