Design Document

Windesheim

Spark! Living Lab Conditioned Goods

Version 1.0

Authors:

Name	**Student number**	**Email**
Clément Perucca	s1177124	s1177124@student.windesheim.nl
Florian Paul	s1177122	s1177122@student.windesheim.nl
Thibaut Ribe	s1177132	s1177132@student.windesheim.nl
Nick van de Poel	s1139131	s1139131@student.windesheim.nl
Mischa Heimenberg	s1111178	s1111178@student.windesheim.nl

Version history

Version	When	Who	What
0.0.1	2021-10-29	Thibaut Ribe, Nick van der Poel and Florian Paul	Write initial improved blockchain lifecycle.
0.0.2	2021-11-02	Florian Paul	Table of function access v1
0.0.3	2021-11-12	Nick van der Poel, Thibaut Ribe	Write initial set of functions
0.0.4	2021-11-19	Nick van der Poel	Updated chaincode functions
0.0.5	2021-11-24	Clement Perucca, Florian Paul	Couple chaincode functions to requirements
0.0.6	2021-11-24	Florian Paul	Table of function access v2
0.1.0	2022-01-07	Nick van der Poel	Added BPMN
1.0	2022-01-10	All Authors	Final version

Distribution management

Version	When	Who
1.0	2022-01-14	Windesheim

Table of contents

1. Introduction

We are a group of students from different nationalities and backgrounds who have come together to work on behalf of the Windesheim Zwolle on a project for the external organisation Spark! Living Lab.

We are all attending the security engineering minor at Windesheim Zwolle through which we have gotten the opportunity to work with Spark! Living Lab on their “Conditioned goods” project. We are the third iteration of developers working on the blockchain project hoping to deliver an application of blockchain with data-assurance that is usable for Spark’s client LambWeston. We define data-assurance as the process(es) we use to increase the trustability of the data-collected by our application[1][2].

Spark! Living Lab is a research company focused on applying IoT and blockchain to the (cold) supply chain industry. Spark accepts projects from various companies depending on a (cold) supply chain. Spark’s goal is to try and figure out how their clients can benefit from IoT and/or blockchain. Currently a lot of companies are experimenting with applying blockchain and/or IoT and are looking for a way to adopt these technologies, that’s where Spark steps in.

LambWeston-Meijer (LWM) is one of those clients looking to apply IoT and blockchain to their cold chain. LWM is a large manufacturer of french fries and other processed potato products. They distribute their goods all over Europe using coldchain. In this kind of chain, the produced goods are kept in controlled environments targeting cold preservation of goods. From the moment they are being processed in the factory until the moment they are stored in the customers’ fridge, the temperature is to be kept within pre-agreed boundaries. Something that is monitored manually at the moment could be automated.

In this document we will follow up on our previous analysis report and the requirement traceability matrix (RTM) bundled with it. This document aims to present a more technical description of the Hyperledger Fabric network for LambWeston’s case and how to build it. We started designing our product using the previous group’s Hyperledger Fabric network which is what we start chapter 2 off with. Right after that we discuss logging temperature onto the blockchain. Followed up by an illustration of how the chaincode lifecycle would work from beginning to end backed up by a BPMN 2.0. After that an implementable interface of functions and how they relate to the earlier collected requirements. Finally we will finish the document using an overview of who should be able to access which functionalities.

2. The shipment lifecycle

Based on the previous chaincode the following concepts and ideas have been introduced:

First, adding a notion of ownership to the shipments. This new model of ownership aims to remove the dependency on external applications to know who would have been the owner at time of the SLA-breach. So LambWeston would reliably know who to contact.

Second, enforcing strict relations between shipments and sensors. One sensor can only be assigned to one shipment at a time and in order to assign it to another sensor it has first to be unregistered from the shipment.

2.1 Logging temperatures on the blockchain

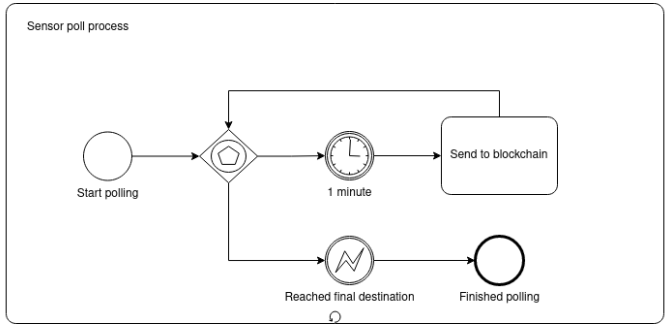
How readings are inserted is heavily dependent on how the sensors themselves work. We were not able to get a proper understanding of how the sensors provided by InnoTractor would have workedn as we have not received them. For that reason we do not describe a way to do so. However we will assume that some external application would invoke the **addMeasurement** function.

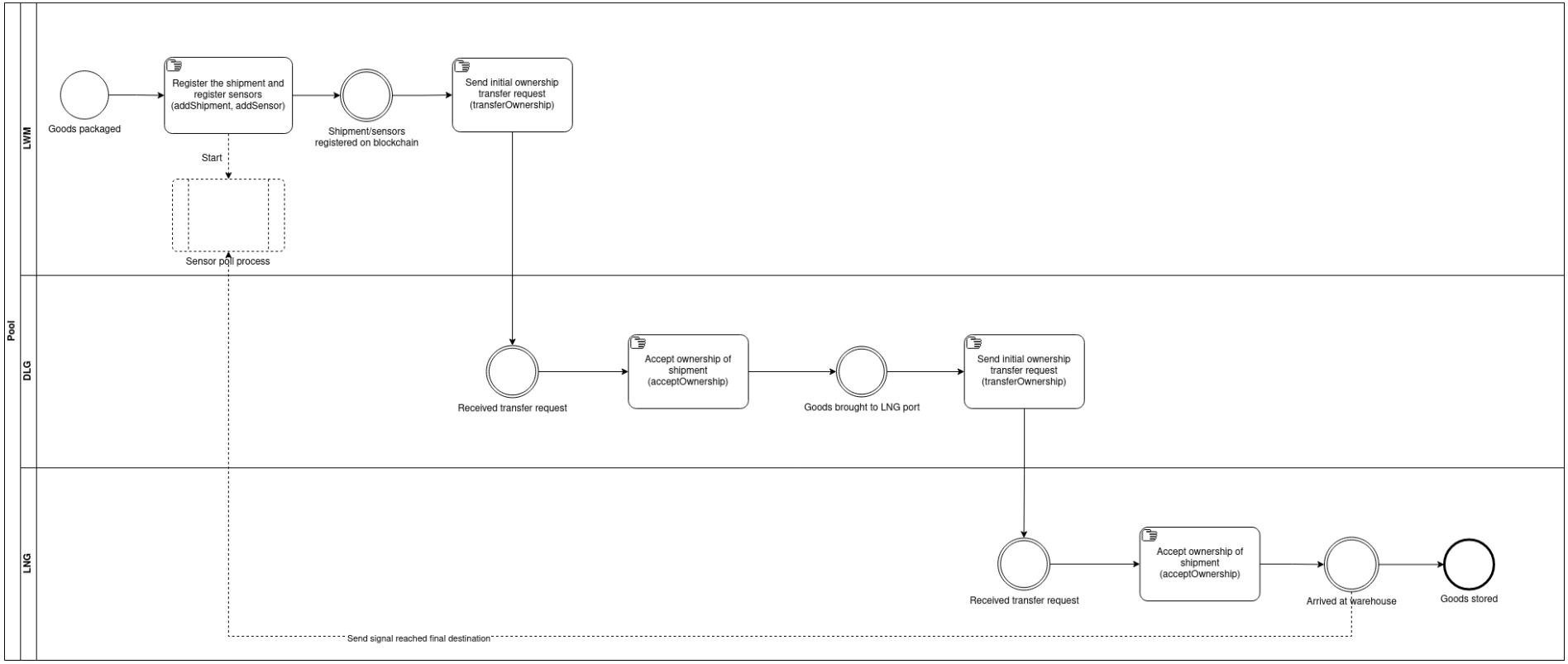
2.2 From beginning to end

Starting after the fresh processed potato products roll off the band and they get packed into containers (**addShipment**). Once the sensors are placed on the shipment they will be initialised (**registerSensor**) and registered if they weren't already (**createSensor**). From the moment the sensors are registered on the shipments the sensors will continuously monitor the temperature and GPS on an interval of 1 minute and be inserted into the blockchain by the LambWeston organisation (**addMeasurement**). One sensor can only be connected to one shipment. The life cycle then gets kicked off by an initial transfer of responsibility from LambWeston to DLG (**transferOwnership, acceptOwnership**). After arriving at the port another transfer of responsibility will be done by DLG to Lineage (**transferOwnership, acceptOwnership**).

2.3 BPMN

This BPMN on the following page is used to visualise the life cycle described in chapter 2.2. The BPMN below has been made using BPMN 2.0.


Sensor poll process
Start polling
1 minute
Send to blockchain
Reached final destination
Finished polling


Pool
LWM
DLG
LNG
Goods packaged
Register the shipment and register sensors (addShipment, addSensor)
Shipment/sensors registered on blockchain
Send initial ownership transfer request (transferOwnership)
Start
Sensor poll process
Received transfer request
Accept ownership of shipment (acceptOwnership)
Goods brought to LNG port
Send initial ownership transfer request (transferOwnership)
Received transfer request
Accept ownership of shipment (acceptOwnership)
Arrived at warehouse
Goods stored
Send signal reached final destination

3. Chaincode interface

The chapter is to serve as an easy implementable interface for the smart contracts. Functions prefixed by the `public` keyword should be invokable as part of the smart contracts. Functions prefixed with the keyword `private` act as guidelines for implementing reusable functions. Function signatures prefixed with a `?` could be useful but is not related to data-assurance.

3.1 Types

```
type Shipment = {
 /**
  * A unique identifier for every shipment
  */
 id: string;
 /**
  * A set of unique identifiers that can be used to fetch sensors
  */
 sensorIds: string[];
 /**
  * The unix epoch at which the shipment was created
  */
 createdAt: number;
 /**
  * The current party having ownership over the shipment must be an
existing MSPID
  */
 owner: string;
}

enum ESensorType {
 Temperature
}
```

```
type Sensor = {
 /**
  * A unique identifier for every shipment
  */
 id: string;
 /**
  * The type of sensor
  */
 category: ESensorType;
 /**
  * The last reading of the sensor
  */
 value: string;
 /**
  * The unix epoch at which the last time the value was updated
  */
 updatedAt: number;

}



enum EShipmentTransferState {
   Pending,
   Cancelled,
   Completed
}
type ShipmentTransfer = {
   /**
    * MSPID from who made the transfer request
    */
   fromOwner: string,//from who the transfer should be send to
   toOwner: string,
   /**
    * The identifier of the shipment in question
    */
   shipmentId: string,
   /**
    * The state of the transfer request
    */
   state: EShipmentTransferState
}
```

3.2 Shipments

public addShipment: (ctx: Context , id: string, createdAt: number) => Promise<void>;
Used for creating a new shipment with the supplied `id` as a unique identifier. Any naming scheme can be used for `id` as long as it is unique. This function will insert a new `Shipment` instance into the state database (CouchDB). Although not something to be wished, `createdAt` would have to be supplied as a separate argument as fetching the current unix epoch is nondeterministic and can result in an unsynced state database. The owner field of the shipment is by default set to the MSPID of the party that invokes this function.

The unique identifier is generated by using the method sensorExist() to check if the Id is already used or not.

N.	Requirement	How does it complete the requirement
11	There cannot be shipments and sensors with duplicate identifiers in the system	The unique identifier is checked by using the method shipmentExist() to ensure that it has not been used for another shipment.

public transferOwnership: (ctx: Context, id: string, newOwner: string) => Promise<void>;
Used for initiating a transfer of ownership which will have to be accepted by the newOwner party as well. In order to invoke this function your MSPID needs to match the one of the corresponding `Shipment` instances. Invoking this function also results in a new `ShipmentTransfer` instance being created and persisted to the state database. The state of the `ShipmentTransfer` is `Pending` by default. This 2-way agreement will prevent unagreed upon transfers of `Shipment` instances. With this we can guarantee that if the SLA is breached that we can trace back who was responsible at that time. The identifier of the transfer request is also unique.

N.	Requirement	How does it complete the requirement
14	The smart contracts must allow for transfer of ownership of shipments	It allows the initiation of a ownership transfer.
4	Unauthorised users cannot update the blockchain	Check if the MSPID matches the one of the corresponding `Shipment` instances

public acceptOwnership: (ctx: Context, id: string) => Promise<boolean>;
Used for marking a `ShipmentTransfer` instance as completed. This under the condition that you carry the MSPID of the `toOwner` field and that such a `ShipmentTransfer` instance exists. Once accepting this offer there is no way to undo the transfer besides invoking a new `transferOwnership` contract. It will return a boolean to indicate whether the transfer was successful.

N.	Requirement	How does it complete the requirement

14	The smart contracts must allow for transfer of ownership of shipments	This function allows for the finalisation of a transfer request.
4	Unauthorised users cannot update the blockchain	Check if the MSPID matches the one of the corresponding `Shipment` instances

?public listOwnershipOffers: (ctx: Context) => Promise<ShipmentTransfer[]>; (not implemented)

Used for listing all possible ownership offers. You cannot accept what you are not aware of hence this function. This function will use your own MSPID as a starting point for looking up the existing `ShipmentTransfer` instances by getting all instances that have a `toOwner` matching yours.

?public getShipment: (ctx: Context, id: string) => Promise<Shipment | null>;

Used for retrieving a single `Shipment` instance by looking for an instance with a matching identifier as supplied by `id`. Optionally an additional `sensor` field can be returned within the `Shipment` object which holds all `Sensor` instances (sensor readings etc.) based on the `sensorIds` field. With this an additional **getReadings** call could be prevented. The response can be null if a `Shipment` instance with the specified identifier does not exist.

?public shipmentExist: (ctx: Context, id: string) => Promise<boolean>;

Used for simply checking whether a shipment with the given identifier exists. This can be used by the sensors to possibly determine when their job as sensors is complete. There is no constraint to fetching whether a `Shipment` instance with the given name exists.

public registerSensor: (ctx: Context, shipmentId: string, sensorId: string, sensorType: ESensorType) => Promise<void>;
User for registering a new sensor on an existing `Shipment` instance. The conditions for invoking function is that the supplied `sensorId` argument is unique. With the supplied arguments it will try to persist a new sensor instance to the state database. After that it will add the sensor identifier to the `Shipment` instance in the `sensorIds` field. The problem with this approach is that sensors could magically appear. To prevent this, sensors will be taken from a pool of sensors registered and stored in couchDB.

N.	Requirement	How does it complete the requirement
10	Only verified sensors can be applied to shipments	Sensors will be taken from a pool of sensors registered and stored in couchDB.
11	There cannot be shipments and sensors with duplicate identifiers in the system	The conditions for invoking this function is that the supplied `sensorId` argument is unique.

public getReadings: (ctx: Context, shipmentId: string) => Promise<Sensor[]>;
Used for returning all up-to-date states of the registered sensors on the specified shipment.

?public getShipmentBySensor: (ctx: Context, sensorId: string) => Promise<Shipment | null>; (not implemented)
Optionally used for looking up the `Shipment` instance that the given `sensorId` lives on. The result can be null. With the current data structure this is a fairly inefficient/hacky lookup. The sensor should probably have a `shipment` property that holds a reference to the parent `Shipment` instance.

?public getShipments: (ctx: Context, offset: string, amount: string, searchQuery: string | null) => Promise<Shipment[]>; (not implemented)
Optionally used for retrieving a list of shipments that matches the searchQuery argument.

public createSensor: (ctx : Context, category: EsensorType) => Promise<void>;
This function is used to create a sensor and add it to the pool of sensors stored in the state database (CouchDB). The created sensors will have a type (temperature, GPS …) and a unique identifier verified by calling sensorExist().

public getSensor: (ctx: Context, sensorId: string) => Promise<Sensor | null>;
Used for retrieving a single `Sensor` instance by looking for an instance with a matching identifier as supplied by `sensorId`. The response can be null if a `Sensor` instance with the specified identifier does not exist.

private validateSLA: (ctx: Context, sensor: Sensor[]) =>boolean;
Used for verifying that there is no SLA-breach. Returns true if all sensors within the list conform to the SLA.

public sensorExist(ctx: Context, sensorID: string) => Promise<boolean>;
Used to check if a sensor already exists with this ID, return a boolean.

N.	Requirement	How does it complete the requirement
18	Only measurements from verified sensors are inserted into the blockchain	Allows to verify if sensor already exists and has been verified.

public addMeasurement: (ctx: Context, sensorID: string, value: string, timestamp: number) => Promise<void>;
Used for updating a registered `Sensor` instance's reading. This function should theoretically only have to be invoked by sensors.

4. Table of function access

This chapter aims to show for each function which stakeholder should be able to use it. Access control is an important part of security and assuring who has which right will allow trust among the peers.

Measurement:

Functions/Companies	Lamb Weston	DLG	LineAge
addMeasurement		X	
validateSLA	X	X	X

Ownership:

Functions/Companies	Lamb Weston	DLG	LineAge
transferOwnership	X	X	X
acceptOwnership	X	X	X
listOwnershipOffers	X	X	X

Shipments and sensors:

Functions/Companies	Lamb Weston	DLG	LineAge
addShipment	X		
getShipment	X	X	X
shipmentExist	X	X	X
registerSensor	X		
getReadings	X	X	X
getShipmentBySensor	X	X	X
sensorExist	X	X	X
createSensor	X		
getSensor	X	X	X

5. References

[1] Bloomberg, LP. (2021, July 26). How does data assurance increase confidence in data? – The ODI. Retrieved November, 12, 2021, from https://theodi.org/article/how-does-data-assurance-increase-confidence-in-data/

[2] Tatum, M. (n.d.). What is Data Quality Assurance? EasyTechJunkie. Retrieved November, 12, 2021, from https://www.easytechjunkie.com/what-is-data-quality-assurance.htm