UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Análisis de plataformas blockchain para casos de uso de compra/venta de imágenes

por

David Vilaseca Barceló

Director: Jaime Delgado

Departamento: Arquitectura de Computadores

Barcelona, Mayo 2019

Abstract

In this project a study of different blockchain technologies with different design purposes is carried out.

The chosen technologies are Ethereum and Hyperledger, of which there will be a basic explanation of their operation and their main characteristics as development platforms.

Some use cases applied to the purchase / sale of images using one of the technologies for each of the cases will be considered.

One of those cases will be chosen and the development of a demo will be deepened using the tools provided by that platform, in addition to those that are necessary to carry it out.

Finally, the pros and cons of each technology will be compared for the scenario we propose and we will proceed to draw conclusions in this regard.

Resum

En aquest projecte es realitza un estudi de diferents tecnologies blockchain amb diferents propòsits de disseny.

Les tecnologies escollides són Ethereum i Hyperledger, de les quals es farà una explicació bàsica del seu funcionament i de les seves característiques principals com a plataformes de desenvolupament.

Es plantejaran alguns casos d'ús aplicats a la compra / venda d'imatges usant una de les tecnologies per cada un dels casos.

S'escollirà un d'aquests casos i s'aprofundirà en el desenvolupament d'una demo usant les eines proporcionades per aquesta plataforma, a més de les que es vegin necessàries per dur-la a terme.

Finalment es compararan els pros i els contres de cada tecnologia per a l'escenari que plantegem i es procedirà a realitzar unes conclusions al respecte.

Resumen

En este proyecto se realiza un estudio de diferentes tecnologías blockchain con distintos propósitos de diseño.

Las tecnologías escogidas son Ethereum e Hyperledger, de las cuales se hará una explicación básica de su funcionamiento y de sus características principales como plataformas de desarrollo.

Se plantearán algunos casos de uso aplicados a la compra/venta de imágenes usando una de las tecnologías por cada uno de los casos.

Se escogerá uno de esos casos y se profundizará en el desarrollo de una demo usando las herramientas proporcionadas por esa plataforma, además de las que se vean necesarias para llevarla a cabo.

Finalmente se compararán los pros y los contras de cada tecnología para el escenario que planteamos y se procederá a realizar unas conclusiones al respecto.

Reconocimientos

Mi agradecimiento y mi cariño a todos los compañeros y amigos que he ido conociendo a lo largo de la carrera, la cual me ha hecho crecer y mejorar como persona.

También agradecer a Jaime Delgado y a Silvia Llorente por la supervisión y consejos que me han dado a lo largo del desarrollo del proyecto.

Índice

Índice de figuras

1. Introducción

La tecnología blockchain ha tenido mucho protagonismo en los últimos años, sobretodo por su asociación con las conocidas criptomonedas. Pero su aplicación práctica va más allá de eso, y con el tiempo se han ido desarrollando algunas alternativas que se apartan de éste uso típico.

Usando blockchain, queremos plantear una solución a la poca trazabilidad que tienen las compraventas de imágenes, vinculando de algún modo las imágenes a la blockchain y aprovecharnos de las característica de ésta para almacenar registros inmutables de transacciones.

Dado que cada tecnología blockchain tiene un enfoque distinto en cuanto a la implementación de ciertas características de la blockchain que implementan, la intención de este proyecto es el de estudiarlas y poder justificar el uso de blockchain en un escenario de tipo B2B. Las tecnologías que usaremos son Ethereum e Hyperledger Fabric.

Originalmente se pretendía usar una herramienta para la manipulación de los metadatos de imágenes JPEG, pero por dificultades en el uso de las tecnologías blockchain se acabó desestimando su uso.

2. Estado del arte de la tecnología usada

La tecnología blockchain consiste en que cada nodo de la red almacena una copia exacta de las transacciones realizadas en ella. Hay distintas plataformas que implementan esta idea. Estos son algunos ejemplos de ellas:

- Ethereum
- Hyperledger
- Openchain
- R3 Corda

Este proyecto ha escogido Ethereum porque ya se tenía experiencia previa con esta plataforma y cuenta con versiones estables, e Hyperledger Fabric porque también cuenta con versiones estables disponibles y la opción de poder usar distintos lenguajes para poder implementar la solución, entre ellos Javascript, Python y Go.

En este apartado se explican brevemente las tecnologías blockchain escogidas que se van a usar en el proyecto y las herramientas que son necesarias para su funcionamiento.

2.1. Ethereum

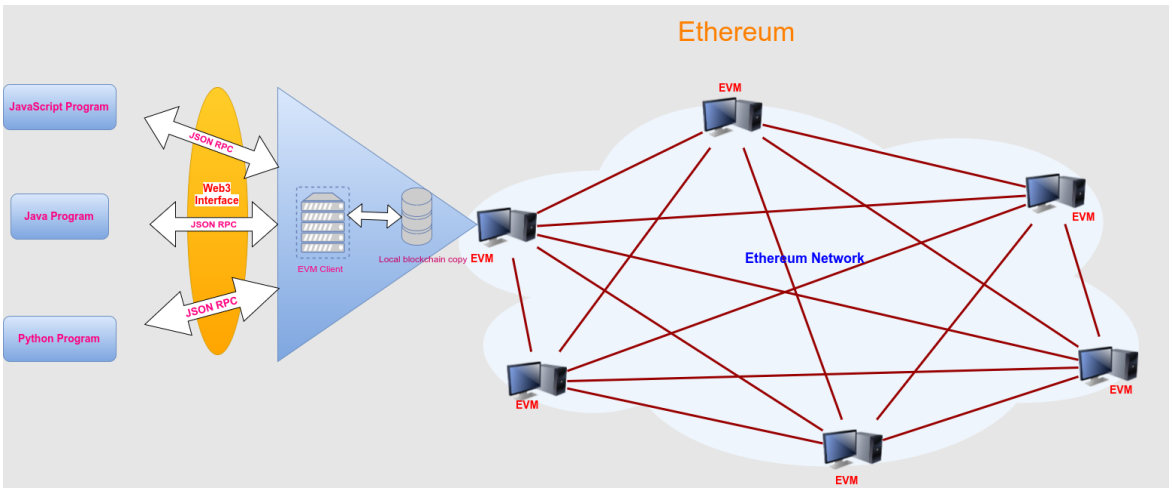

Imagen 1: Esquema de la comunicación con una red Ethereum

Ethereum es una red de blockchain pública distribuida de código abierto para desarrollar aplicaciones descentralizadas utilizando a lo que se conoce como Smart Contract.

Al ser pública, cualquier persona puede formar parte de ella sin requerir de permiso de ningún tipo.

Los Smart Contracts son pequeñas piezas de código desarrollados principalmente en el lenguaje Solidity, que se ejecutan en cada uno de los nodos que formen parte de la red y las que se les asocia una cuenta de la red.

El funcionamiento de Ethereum está vinculado a su propia criptomoneda llamada Ether, la cual se genera “minando” bloques de la red y recompensando al nodo que lo consiga primero.

La “minería” de bloques en Ethereum se basa en el concepto de Proof of Work (PoW). Un bloque se considera minado cuando un nodo de la red encuentra solución a un algoritmo diseñado específicamente para la plataforma llamado Ethash.

Cuando se encuentra una posible solución, el nodo que la ha encontrado propone esta solución a los demás nodos, que comprobaran si esa solución es la correcta de manera democrática. Como recompensa al nodo, se le darán una cantidad de Ethers, que se podrán usar para transferirse a otras direcciones dentro de la red.

Cada nodo puede tener varias cuentas de Ethereum, por lo que se pueden realizar transacciones de Ether entre transacciones. Partiendo de esto, si se transfiere Ether a una cuenta vinculada a un Smart Contract, este se ejecutará.

Dentro del contexto de los Smart Contracts, existe un concepto que limita también el desarrollo de éstos que se conoce como Gas. El Gas actúa como moneda de cambio para ejecutar código en la red, por lo que cuánto más código tenga un Smart Contract, más Ethers costará ejecutarlo. También sirve como limitador de definición del propio Smart Contract, ya que cada nodo deberá ejecutar ese código de manera local y transmitir a la red su resultado, por lo que muchas líneas de código supondría largas esperas para la ejecución del Smart Contract en la red.

Para poder comunicarse con la blockchain de Ethereum, se nos proporciona una librería llamada Web3 la cual está disponible en varios lenguajes, entre ellos Javascript que es la que se va a usar.

2.2. Hyperledger Fabric

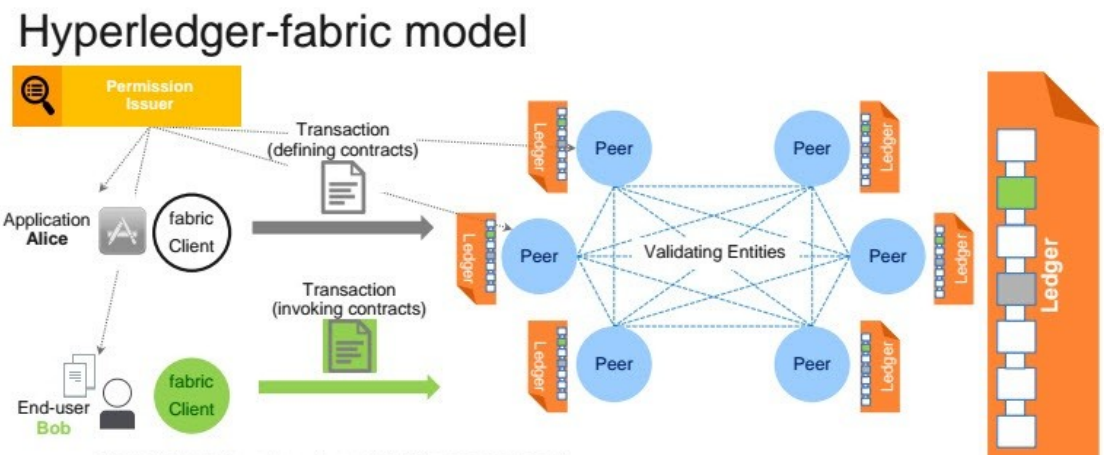

Imagen 2: Esquema comunicación con una red Hyperledger Fabric

Hyperledger Fabric ha sido diseñada para desarrollar aplicaciones blockchain con la flexibilidad de los permisos y una alta escalabilidad. Además, sigue una arquitectura modular, por lo que se pueden añadir algoritmos específicos para la identidad o la encriptación.

Proporciona también una eficiencia de procesado debida una asignación por roles según el tipo de nodo. De esta manera se añade concurrencia a la red a la hora de realizar los distintos pasos para procesar una transacción.

Otro elemento que distingue Hyperledger Fabric de las redes públicas es el concepto de canal, que básicamente serían blockchains dentro de la propia blockchain, añadiendo una capa de privacidad y confidencialidad.

Al contrario que en las redes públicas, que permiten que cualquier identidad desconocida forme parte de la red, los miembros deben enrolarse mediante el MSP, el cual es una interfaz que permite abstraerse de los mecanismos criptográficos y de los protocolos

detrás los distintos procesos de autentificación. Esta interfaz puede implementarse de distintas maneras, como por ejemplo usando el llamado Fabric CA's que es la implementación por defecto de la interfaz MSP al que se le pueden hacer peticiones de membresía, o directamente se pueden generar los certificados mediante una herramienta llamada cryptogen, que facilita la rápida implementación de un MSP para un rápido desarrollo.

En realidad, el enrolamiento funciona mapeando el MSP a lo que se conoce como Organizaciones. Cada peer debe formar parte de alguna organización, y estas pueden tener múltiples peers como miembros.

Tal y como se ha comentado, en Hyperledger Fabric hay varios tipos de nodos (o peers) dependiendo de su rol asignado. Hay tres tipos de roles distintos:

- Commiting peer: Realiza transacciones y mantiene el estado de la blockchain.
- Endorsing peer: Recibe propuestas de transacciones para aprobarlas, respondiendo con o sin la aprobación.
- Ordering peer: Aprueba la inclusión de bloques de transacciones en la blockchain y se comunica con los otros tipos de peers.

Hyperledger Fabric dispone también del concepto de Smart Contracts. En su caso, ya tiene soporte oficial en varios lenguajes de programación como Go, Python y Nodejs.

3. Casos de uso

3.1. Registro de transacciones entre distribuidor y comprador

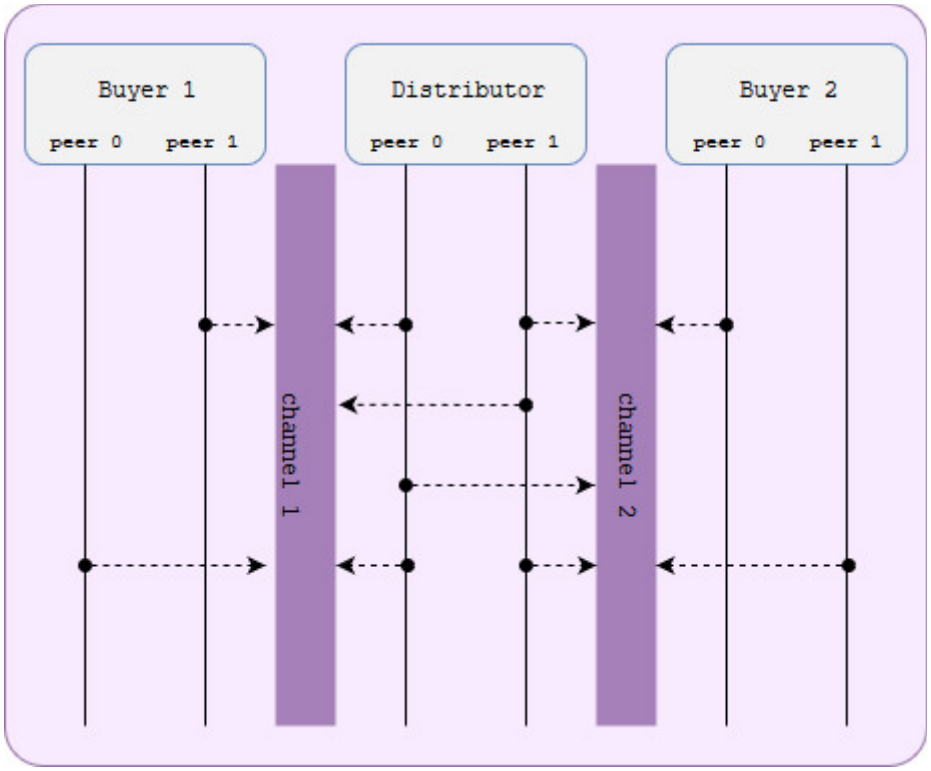

Imagen 3: Diagrama funcionamiento del caso de uso distribuidor-comprador

Consideremos un distribuidor B que desea vender una serie de imágenes y un comprador C desea comprarlas. El Smart Contract registrará los datos de las ventas en la blockchain.

Un usuario realiza una petición para consultar las transacciones realizadas entre el distribuidor B y el comprador C. El Smart Contract recupera el historial de esas transacciones ligadas a distintas imágenes.

3.2. Registro de transacciones por imagen

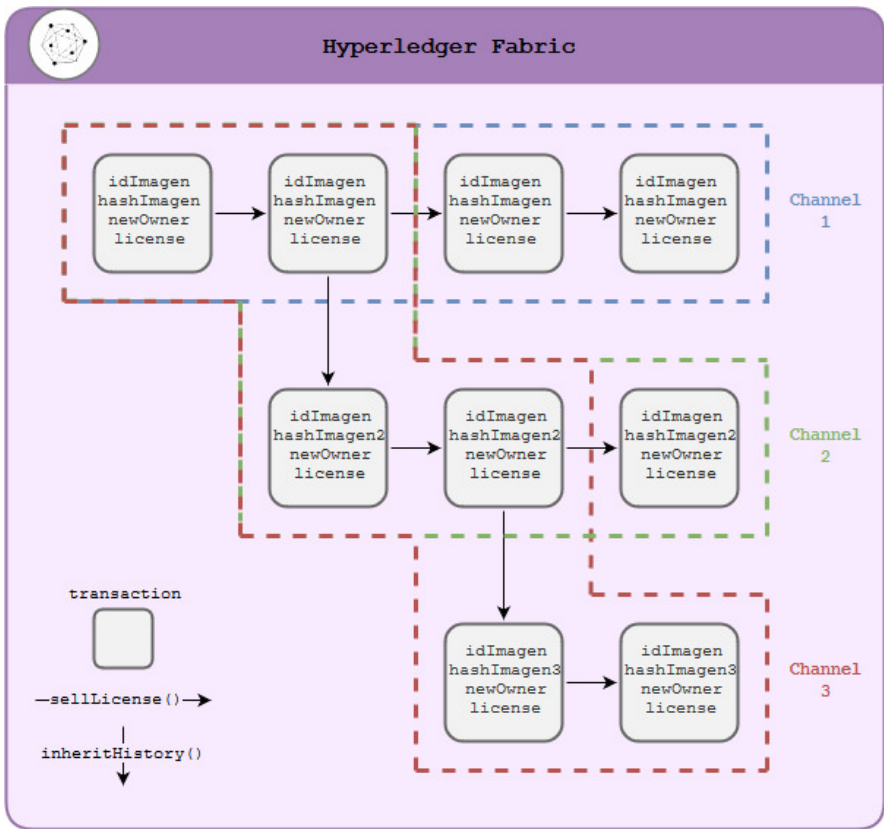

Imagen 4: Diagrama funcionamiento del caso de uso por imágenes

Consideremos un vendedor B que desea vender una imagen F y un comprador C desea comprarla. El Smart Contract registrará los datos de la venta de la imagen en la blockchain.

Acto seguido, el vendedor C desea vender la imagen F y un comprador D desea comprarla. De nuevo el Smart Contract almacenará los datos de esa transacción.

Un usuario realiza una petición para consultar las transacciones realizadas sobre la imagen F. El Smart Contract recupera el historial de todas las transacciones hechas ligadas a esa imagen.

4. Metodología y desarrollo del proyecto

4.1. Requisitos

Se desea implementar un servicio que nos comunique con una red blockchain en la cual se ha desplegado un Smart Contract que cumpla con el caso de uso aplicado. Para consumir este servicio se quiere una aplicación web con la que poder realizar ventas de imágenes asociadas a las licencias de uso y poder consultar el historial de transacciones de estas.

4.2. Herramientas utilizadas durante el desarrollo

Para desarrollar este proyecto se ha utilizado el sistema operativo Ubuntu 14.04 LTS y se ha utilizado el editor de texto Visual Studio Code para definir los ficheros de configuración necesarios y para desarrollar el software.

Para cada implementación desarrollada se ha creado un repositorio usando la herramienta de control de código fuente Git.

Para la aplicación web se ha usado Angular, el cual está diseñado pensando en la arquitectura MVC. Se basa en una estructura de clases tipo Componente, cuyas propiedades se usan para vincularse con los datos del documento HTML.

4.3. Herramientas de desarrollo de Ethereum

Cada nodo de la red necesita un cliente de Ethereum para poder hacer peticiones a la red blockchain. En nuestro caso usamos el cliente Geth el cual usaremos a través de una librería llamada web3.js.

Para desarrollar el Smart Contract usaremos un IDE del lenguaje Solidity vía navegador web llamado Remix.

4.4. Herramientas de desarrollo de Hyperledger Fabric

Hyperledger Fabric proporciona una serie de herramientas para llevar a cabo y facilitar el desarrollo del software.

Un SDK para poder que nos permite realizar las peticiones de transacción y consulta a la blockchain.Hay 3 SDKs oficiales correspondientes a los lenguajes Go, Python y Node.js, de los cuales se usará el de Node.js.

También es necesario el uso de Docker para poder generar contenedores de las entidades que forman parte de la blockchain, como los peer o el orderer.

Hay otras herramientas proporcionadas que su objetivo es puramente el de facilitar un desarrollo rápido pero que no se recomienda su uso en un entorno de producción. Su función se explicará en el apartado de implementación que sigue.

4.5. Implementación

4.5.1. Ficheros de configuración

Tal y como se ha explicado, también se usa la tecnología docker para instanciar cada uno de los participantes necesarios para el funcionamiento de una red Hyperledger Fabric.

Lo primero que debemos hacer es definir una serie de ficheros de configuración que necesita Hyperledger Fabric para funcionar del modo que queremos. Estos ficheros son los siguientes:

- configtx.yaml
- crypto-config.yaml
- docker-compose.yaml

El fichero configtx.yaml nos permite definir el nombre de las distintas organizaciones que formarán parte de nuestra red, los puertos por los cuales se comunicarán los distintos participantes y el nombre del consenso que se utilizará. En nuestro caso solo queremos una organización, de la cual formarán parte todos los peers. Además, el servicio conocido como Orderer forma parte de una organización propia, independiente de la que los peers forman parte. El motivo de esto es que el orderer podría dar servicio a otras organizaciones de la blockchain si se diese el caso.

Una vez definido el fichero, usamos el binario llamado configtxgen para que consuma el fichero y cree los ficheros de configuración de transacción del bloque genesis del orderer, los de los posibles canales que se vayan a usar y los de los peers..

Cada uno de estos ficheros se crean usando unos flags en específico del binario, y que se crean de antemano ya que trabajaremos en un entorno de desarrollo.

El fichero crypto-config.yaml nos permite definir el dominio de las organizaciones de la red y cuántos integrantes tendrá cada una de ellas. En nuestro caso tendremos 5 peers que formarán parte de la organización AppMSP.

Seguidamente se usará el binario cryptogen para que consuma éste fichero y genere el material criptográfico correspondiente a cada uno de los participantes de la red, el orderer y los 5 peers. Los participantes usarán éste material para certificar su identidad dentro de la red y poder realizar distintas peticiones y transacciones.

Por último, tenemos el fichero docker-compose.yaml, que es el que necesita docker para instanciar los contenedores generados a partir de las imágenes proporcionadas por Hyperledger para cada uno de los servicios. En nuestro caso definiremos que nos son necesarios 5 contenedores para los peers, 1 contenedor para el orderer y 1 para el contenedor del cli, el cual podrá usarse como mediador para poder ejecutar comandos o ver registros de trazas en el resto de contenedores.

Una vez les hemos dado uso a estos ficheros, ya tenemos el entorno preparado para poder empezar a desarrollar el software.

4.5.2. Interfaz de usuario

El front-end consta de una SPA desarrollada en Angular tal y como hemos dicho antes.

Podemos ver dos botones, tres desplegables y un checkbox.

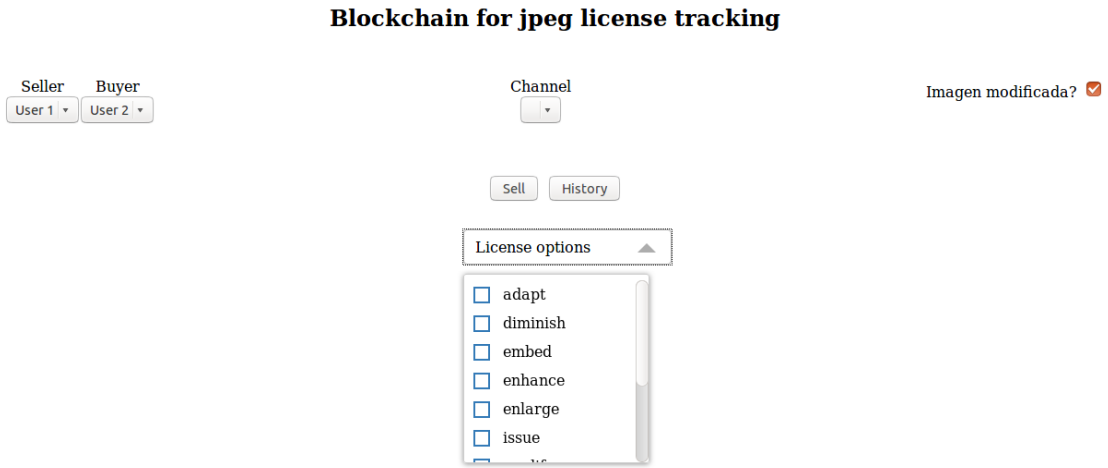

Imagen 5: Vista front-end

El botón Sell es el que usaremos para que el vendedor tramite la venta al comprador. Esta venta estará asociada a los parámetros seleccionados de la licencia de uso de la imagen en el desplegable central inferior. Todo esto será registrado en el canal seleccionado en el desplegable central superior.

En caso de tener marcado el checkbox, simularemos que se ha alterado la imagen, por lo que el botón Sell creará un canal nuevo para registrar la nueva venta.

El botón History recuperará el historial de ventas asociado a la imagen, en función de qué canal tengamos seleccionado. El listado de canales dependerá de qué usuario tengamos seleccionado como vendedor.

idImage	hashImage	newOwner	license									
			adapt	diminish	embed	enhance	enlarge	issue	modify	play	print	reduce
stonehenge	081fc7c055dad3ae8ac58619b9e9eff3b22d9d22dfbdb142b2097074c06d1daf	user2	false	false	false	false	false	false	false	false	false	false
stonehenge	081fc7c055dad3ae8ac58619b9e9eff3b22d9d22dfbdb142b2097074c06d1daf	user3	true	false	false	false	false	false	false	false	false	false
stonehenge	081fc7c055dad3ae8ac58619b9e9eff3b22d9d22dfbdb142b2097074c06d1daf	user2	false	false	false	false	false	false	false	false	false	false
stonehenge	3e919848e1e346a9ce98d1768d9eb409fa799a3ee9fb1a7dcaabebcd4e75610c	user1	false	false	false	false	false	false	false	false	false	false

Imagen 6: Tabla del historial de la imagen

Se ha considerado suficiente definir solamente el componente blockchain.component.ts para comunicarse con la vista, ya que no es objetivo principal del proyecto y ha agilizado la capacidad de poder testear la lógica implementada en el proyecto.

Además, se han creado dos componentes de tipo servicio que serán los encargados de realizar las peticiones al servidor:

- FabricService
- HashService

El componente fabric.service.ts se encarga de realizar las peticiones que están destinadas a comunicarse con la red de blockchain, tanto peticiones de información como transacciones.

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { Observable } from 'rxjs';
import { SellLicenseRequest } from '../interfaces/sellLicenseRequest';
import { GetHistoryRequest } from '../interfaces/getHistoryRequest';
import { User } from '../interfaces/user';
import { InheritHistoryRequest } from
'../interfaces/inheritHistoryRequest';

@Injectable({
   providedIn: 'root'
})
export class FabricService {

   private queryUrl = 'http://localhost:3000/api/getHistory';
   private transactionUrl = 'http://localhost:3000/api/sellLicense'
   private getChannelsUrl = 'http://localhost:3000/api/getChannels';
   private inheritHistoryUrl =
'http://localhost:3000/api/inheritHistory';

   constructor(private http: HttpClient) { }
   sellLicense(request: SellLicenseRequest): Observable<any> {
       const headers = new HttpHeaders()
           .set('Content-Type', 'application/json');
       return this.http.post(this.transactionUrl,
JSON.stringify(request), {
           headers: headers
       })
   }
   getHistory(request: GetHistoryRequest): Observable<any> {
       const headers = new HttpHeaders()
           .set('Content-Type', 'application/json');
       return this.http.get(this.queryUrl, {
           headers: headers,
           params: {
               request: JSON.stringify(request)
           }
       });
   }
```

```
    getChannels(user: User): Observable<any> {
        const headers = new HttpHeaders()
            .set('Content-Type', 'application/json');
        return this.http.get(this.getChannelsUrl, {
            headers: headers,
            params: {
                user: JSON.stringify(user)
            }
        });
    }

    inheritHistory(request: InheritHistoryRequest): Observable<any> {
        const headers = new HttpHeaders()
            .set('Content-Type', 'application/json');
        return this.http.post(this.inheritHistoryUrl,
JSON.stringify(request), {
            headers: headers
        });
    }
}
```

Imagen 7: Código del servicio FabricService

El componente hash.service.ts se encarga de realizar las peticiones para que el servidor haga las acciones correspondiente a la imagen.

```
import { Injectable } from "@angular/core";
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { Observable } from "rxjs";

@Injectable({
   providedIn: 'root'
})

export class ImageService {

   private getHashUrl = 'http://localhost:3000/api/getHash';
   private putMetadataUrl = 'http://localhost:3000/api/putMetadata';

   constructor(private http: HttpClient){}

   getHash(): Observable<any> {
       const headers = new HttpHeaders()
           .set('Content-Type', 'text/plain');
       return this.http.get(this.getHashUrl, {
           headers: headers
       });
   }

   putMetadata(channel: string): Observable<any> {
       let body = {channel: channel}
       const headers = new HttpHeaders()
           .set('Content-Type', 'application/json');
       return this.http.post(this.putMetadataUrl, body, {
           headers: headers
       });
   }
}
```

Imagen 8: Código del servicio HashService

También se han definido una serie de modelos de datos que se usarán para realizar las peticiones. Aquí se puede ver el ejemplo de uno de ellos.

```
export class Transaction {
    idImage: string;
    hashImage: string;
    newOwner: string;
    license: License;
}

export class License {
    adapt: boolean;
    diminish: boolean;
    embed: boolean;
    enhance: boolean;
    enlarge: boolean;
    issue: boolean;
    modify: boolean;
    play: boolean;
    print: boolean;
    reduce: boolean;
}
```

Imagen 9: Modelo del tipo Transaction

4.5.3. Servidor

El servidor se ha montado usando la librería Express.js. En el fichero server.js usamos esta librería y definimos los enrutamientos para aceptar cada uno de los tipos de petición que puedan venir de parte del cliente.

El uso del SDK está encapsulado en una serie de funciones definidas en los ficheros invoke.js, query.js y helper.js.

4.5.3.1. Fichero invoke.js

El fichero invoke.js contiene las funciones que realizan cambios en la blockchain en forma de transacciones.

Éstas son:

- joinChannel
- createChannel
- newTransaction

La función joinChannel acepta tres parámetros de entrada:

- ➔ channelID: el canal al que quiere unirse el peer.
- ➔ peerUrl: la url del peer. Ej: grpcs://localhost:7051 .
- ➔ targetPeerId: la id del peer. Ej: peer1.

```
module.exports.joinChannel = function (channelID, peerUrl, targetPeerId) {
    return new Promise((resolve, reject) => {
        let fabric_client = new Fabric_Client();
        let channel = fabric_client.newChannel(channelID)
        let peer = fabric_client.newPeer(peerUrl, {
            pem: Buffer.from(serverCert).toString(),
            "ssl-target-name-override": targetPeerId + '.app.jpeg.com'
        });
        let order = fabric_client.newOrderer('grpcs://localhost:7050', {
            pem: Buffer.from(orderCert).toString(),
            'ssl-target-name-override': 'orderer.jpeg.com'
        })
        helper.loadUser(fabric_client, 'Admin')
            .then(user_from_store => {
                let tx_id = fabric_client.newTransactionID();
                let genesisBlock;
                let ordererRequest = {
                    txId: tx_id,
                    orderer: order
                }
                return channel.getGenesisBlock(ordererRequest).then(block => {
                    genesisBlock = block
                    tx_id = fabric_client.newTransactionID();
                    let joinChannelRequest = {
                        targets: [peer],
                        block: genesisBlock,
                        txId: tx_id
                    }
                    return channel.joinChannel(joinChannelRequest);
                });
            }).then(proposalResponse => {
                if (proposalResponse && proposalResponse[0].response.status == 200) {
                    let status = proposalResponse[0].response.status;
                    resolve(status);
                } else {
                    reject(status);
                }
            }).catch(err => {
                reject(err);
            })
    })
}
```

Imagen 10: Código de la función joinChannel

Esta función añade el peer objetivo a un canal previamente creado en la blockchain.

La función createChannel acepta un parámetro de entrada:

➔ channelID: el nombre del canal que se creará.

```
module.exports.createChannel = function (channelID) {
    return new Promise((resolve, reject) => {
        let fabric_client = new Fabric_Client();
        let order = fabric_client.newOrderer('grpcs://localhost:7050', {
            pem: Buffer.from(orderCert).toString(),
            'ssl-target-name-override': 'orderer.jpeg.com'
        });

        helper.loadUser(fabric_client, 'Admin').then(user_from_store => {
            let envelope = fs.readFileSync(path.join(__dirname, '..', '..', 'channel-
artifacts', channelID + '.tx'))
            let config = fabric_client.extractChannelConfig(envelope);
            let signatures = [];
            let signature = fabric_client.signChannelConfig(config);
            signatures.push(signature);
            let txId = fabric_client.newTransactionID();
            let channelRequest = {
                name: channelID,
                orderer: order,
                config: config,
                signatures: signatures,
                txId: txId
            };
            fabric_client.createChannel(channelRequest).then(response => {
                if (response.status == 'SUCCESS')
                    resolve(response.status);
                else {
                    reject(response.status)
                }
            })
        })
    })
}
```

Imagen 11: Código de la función createChannel

Esta función crea un canal con el nombre especificado en la blockchain.

La función newTransaction acepta cuatro parámetros de entrada:

- ➔ seller: la instancia de tipo User correspondiente al vendedor.
- ➔ buyer: la instancia de tipo User correspondiente al comprador.
- ➔ ch: el nombre del canal donde se hará la transacción.
- ➔ transaction: la instancia de tipo Transaction que se almacenará en el canal.

La función crea un bloque nuevo en el canal especificado dónde se almacena la transacción.

```
module.exports.newTransaction = function (seller, buyer, ch, transaction) {
    return new Promise((resolve, reject) => {
        let fabric_client = new Fabric_Client();
        let channel = fabric_client.newChannel(ch);
        let sellerPeer = fabric_client.newPeer(seller.url, {
            pem: Buffer.from(serverCert).toString(),
            "ssl-target-name-override": seller.peer + '.app.jpeg.com'
        });
        let buyerPeer = fabric_client.newPeer(buyer.url, {
            pem: Buffer.from(serverCert).toString(),
            "ssl-target-name-override": buyer.peer + '.app.jpeg.com'
        });
        channel.addPeer(sellerPeer);
        channel.addPeer(buyerPeer);
        let order = fabric_client.newOrderer('grpcs://localhost:7050', {
            pem: Buffer.from(ordererCert).toString(),
            'ssl-target-name-override': 'orderer.jpeg.com'
        })
        channel.addOrderer(order);
        let store_path = path.join(__dirname, '..', '..', 'crypto-
config/peerOrganizations/app.jpeg.com/ca');
        var tx_id = null;
        helper.loadUser(fabric_client, seller.userName).then(user_from_store => {
            tx_id = fabric_client.newTransactionID();
            var request = {
                targets: [sellerPeer, buyerPeer],
                chaincodeId: 'jpeg',
                fcn: 'NewTransaction',
                args: [transaction, 'stonehenge'],
                chainId: ch,
                txId: tx_id
            };
            console.log(request)            var proposal = results[1];
            let isProposalGood = false;
            return channel.sendTransactionProposal(request);
        }).then((results) => {
            var proposalResponses = results[0];
            if (proposalResponses && proposalResponses[0].response &&
                proposalResponses[0].response.status === 200) {
                isProposalGood = true;
                console.log('Transaction proposal was good');
            } else {
                console.error('Transaction proposal was bad');
            }
            if (isProposalGood) {
```

```
                console.log(util.format(
                    'Successfully sent Proposal and received ProposalResponse: Status -
%s, message - "%s"',
                    proposalResponses[0].response.status,
proposalResponses[0].response.message));
                var request = {
                    proposalResponses: proposalResponses,
                    proposal: proposal,
                    orderer: order,
                    txID: tx_id
                };
                var transaction_id_string = tx_id.getTransactionID();
                var promises = [];
                var sendPromise = channel.sendTransaction(request);
                promises.push(sendPromise);
                let event_hub = channel.newChannelEventHub(sellerPeer);
                let txPromise = new Promise((resolve, reject) => {
                    let handle = setTimeout(() => {
                        event_hub.unregisterTxEvent(transaction_id_string);
                        event_hub.disconnect();
                        resolve({ event_status: 'TIMEOUT' });
                    }, 120000);
                    event_hub.registerTxEvent(transaction_id_string, (tx, code) => {
                        clearTimeout(handle);
                        var return_status = { event_status: code, tx_id:
transaction_id_string };
                        if (code !== 'VALID') {
                            console.error('The transaction was invalid, code = ' + code);
                            resolve(return_status);
                        } else {
                            console.log('The transaction has been committed on peer ' +
event_hub.getPeerAddr());
                            resolve(return_status);
                        }
                    }, (err) => {
                        reject(new Error('There was a problem with the eventhub ::' +
err));
                    },
                        { disconnect: true }
                    );
                    event_hub.connect();
                });
                promises.push(txPromise);
                return Promise.all(promises);
            } else {
                console.error('Failed to send Proposal or receive valid response. Response
null or status is not 200. exiting...');
                throw new Error('Failed to send Proposal or receive valid response.
Response null or status is not 200. exiting...');
            }
        }).then((results) => {
            console.log('Send transaction promise and event listener promise have
completed');
            if (results && results[0] && results[0].status === 'SUCCESS') {
                console.log('Successfully sent transaction to the orderer.');
```

```
            } else {
                console.error('Failed to order the transaction. Error code: ' +
results[0].status);
            }
            if (results && results[1] && results[1].event_status === 'VALID') {
                resolve('Successfully committed the change to the ledger by the peer');
            } else {
                console.log('Transaction failed to be committed to the ledger due to ::' +
results[1].event_status);
            }
        }).catch((err) => {
            console.error('Failed to invoke successfully :: ' + err);
            reject(err);
        });
    })
}
```

Imagen 12: Código de la función newTransaction

4.5.3.2. Fichero query.js

Este fichero contiene las funciones que realizan alguna petición a la blockchain.

Éstas son:

- getImageHistory
- getChannels

La función getImageHistory acepta tres parámetros:

➔ ch: el nombre del canal en el que se quiere hacer la petición.
➔ querier: la instancia de tipo User correspondiente al usuario que hace la petición.
➔ imagen: el nombre de la imagen.

```
module.exports.getImageHistory = function (ch, querier, imagen) {
    return new Promise((resolve, reject) => {
        let fabric_client = new Fabric_Client();
        let serverCert = fs.readFileSync(path.join(__dirname, '..', '..', 'crypto-
config/peerOrganizations/app.jpeg.com/msp/tlscacerts/tlsca.app.jpeg.com-cert.pem'));
        let channel = fabric_client.newChannel(ch);
        let peer = fabric_client.newPeer(querier.url, {
            pem: Buffer.from(serverCert).toString(),
            "ssl-target-name-override": querier.peer + '.app.jpeg.com'
        });
        channel.addPeer(peer);
       let store_path=path.join(__dirname,'..', '..','crypto-
config/peerOrganizations/app.jpeg.com/ca');
        return helper.loadUser(fabric_client, querier.userName).then((user_from_store) =>
{
            let request = {
                chaincodeId: 'jpeg',
                fcn: 'GetImageHistory',
                args: [imagen]
            };
            return channel.queryByChaincode(request);
        }).then((query_responses) => {
            console.log("Query has completed, checking results");
            if (query_responses && query_responses.length == 1) {
                if (query_responses[0] instanceof Error) {
                    console.error("error from query = ", query_responses[0]);
                } else {
                    console.log("Response is ", query_responses[0].toString());
                    let response = query_responses[0];
                    resolve(response);
                }
            } else {
                console.log("No payloads were returned from query");
            }
        }).catch((err) => {
            console.error('Failed to query successfully :: ' + err);
            reject(err);
        });
    })}
```

Imagen 13: Código de la función getImageHistory

Esta función realiza una petición al canal especificado para obtener el historial de transacciones correspondiente a la imagen.

La función getChannels acepta un parámetro:

- ➔ querier: la instancia de tipo User correspondiente al usuario que hace la petición.

```
module.exports.getChannels = function (querier) {
    return new Promise((resolve, reject) => {
        let fabric_client = new Fabric_Client();
        let serverCert = fs.readFileSync(path.join(__dirname, '..', '..',
'crypto-
config/peerOrganizations/app.jpeg.com/msp/tlscacerts/tlsca.app.jpeg.com-
cert.pem'));
        let peer = fabric_client.newPeer(querier.url, {
            pem: Buffer.from(serverCert).toString(),
            "ssl-target-name-override": querier.peer + '.app.jpeg.com'
        });
        return helper.loadUser(fabric_client,
querier.userName).then((user_from_store) => {
            return fabric_client.queryChannels(peer, false);
        }).then(response => {
            let channels = [];
            response.channels.forEach(function(channel) {
                channels.push(channel.channel_id);
            })
            resolve(channels);
        })
    })
}
```

Imagen 14: Código de la función getChannels

Esta función realiza una petición a la blockchain para obtener la lista de canales en las que el usuario que hace la petición es miembro.

4.5.3.3. Fichero helper.js

Este fichero contiene las funciones más genéricas que encapsulan procesos que son de ayuda a la hora de desarrollar el software.

Éstas son:

- loaduser
- installChaincode
- instantiateChaincode
- hash
- putMetadata
- flatArray

La función loadUser acepta dos parámetros:

➔ fabric_client: es la instancia de la clase Fabric_Client proporcionada por el SDK de Hyperledger Fabric.
➔ user: el nombre del usuario que se quiere cargar.

```
module.exports.loadUser = function (fabric_client, user) {
    return new Promise((resolve, reject) => {
        Fabric_Client.newDefaultKeyValueStore({
            path: store_path
        }).then((state_store) => {
            fabric_client.setStateStore(state_store);
            var crypto_suite = Fabric_Client.newCryptoSuite();
            var crypto_store = Fabric_Client.newCryptoKeyStore({ path: store_path });
            crypto_suite.setCryptoKeyStore(crypto_store);
            fabric_client.setCryptoSuite(crypto_suite);
            let userName = user.charAt(0).toUpperCase() + user.slice(1);
            return fabric_client.getUserContext(userName, true);
        }).then((user_from_store) => {
            if (user_from_store && user_from_store.isEnrolled()) {
                console.log('Successfully loaded ' + user + ' from persistence');
                member_user = user_from_store;
                resolve(user_from_store);
            } else {
                throw new Error('Failed to get ' + user + '.... run registerUser.js');
                reject();
            }
        })
    })
}
```

Imagen 15: Código de la función loadUser

Esta función obtiene las credenciales creadas del usuario para identificarse ante la red y realizar peticiones y/o transacciones.

La función installChaincode acepta tres parámetros:

- ➔ fabric_client: es la instancia de la clase Fabric_Client proporcionada por el SDK de Hyperledger Fabric.
- ➔ peerUrl: la url del peer.
- ➔ targetPeer: el nombre del peer.

```
module.exports.installChaincode = function (fabric_client, peerUrl, targetPeer) {
    return new Promise((resolve, reject) => {
        let peer = fabric_client.newPeer(peerUrl, {
            pem: Buffer.from(serverCert).toString(),
            "ssl-target-name-override": targetPeer + '.app.jpeg.com'
        });
        this.loadUser(fabric_client, 'Admin').then(() => {
            let request = {
                targets: [peer],
                chaincodePath: chaincode_path,
                chaincodeId: 'jpeg',
                chaincodeVersion: 'v1',
                chaincodeType: 'node'
            };
            return fabric_client.installChaincode(request);
        }).then(result => {
            let response = result[0];
            console.log('Chaincode installed in ' + targetPeer)
            resolve(response[0].response.status);
        }).catch(err => {
            reject(err);
        })
    })
}
```

Imagen 16: Código de la función installChaincode

Esta función instala el código del chaincode en el peer específicado.

La función instantiateChaincode acepta tres parámetros:

- ➔ fabric_client: es la instancia de la clase Fabric_Client proporcionada por el SDK de Hyperledger Fabric.
- ➔ users: los usuarios que queremos que sus respectivos peers aprueben la instanciación del chaincode en el canal.
- ➔ ch: el canal en el que se instanciará el chaincode.

```
module.exports.instantiateChaincode = function (fabric_client, users, ch) {
    return new Promise((resolve, reject) => {
        let channel = fabric_client.newChannel(ch);
        let peers = [];
        users.forEach(function(user) {
            let peer = fabric_client.newPeer(user.url, {
                pem: Buffer.from(serverCert).toString(),
                "ssl-target-name-override": user.peer + '.app.jpeg.com'
            });
            peers.push(peer);
            channel.addPeer(peer);
        })
        console.log(peers)
        let order = fabric_client.newOrderer('grpcs://localhost:7050', {
            pem: Buffer.from(orderCert).toString(),
            'ssl-target-name-override': 'orderer.jpeg.com'
        })
        channel.addOrderer(order);
        let txId = null;
        this.loadUser(fabric_client, 'Admin').then(() => {
            txId = fabric_client.newTransactionID();
            let request = {
                targets: peers,
                chaincodeType: 'node',
                chaincodeVersion: 'v1',
                chaincodeId: 'jpeg',
                txId: txId
            }
            return channel.sendInstantiateProposal(request, 120000);
        }).then(results => {
            var proposalResponses = results[0];
            var proposal = results[1];
            console.log(proposalResponses[0]);
            let isProposalGood = false;
            if (proposalResponses && proposalResponses[0].response &&
                proposalResponses[0].response.status === 200) {
                isProposalGood = true;
                console.log('Transaction proposal was good');
            } else {
                console.error('Transaction proposal was bad');
            }
            if (isProposalGood) {
                console.log(util.format(
                    'Successfully sent Proposal and received ProposalResponse: Status
- %s, message - "%s"',
                    proposalResponses[0].response.status,
proposalResponses[0].response.message));
```

```
                var request = {
                    proposalResponses: proposalResponses,
                    proposal: proposal,
                    orderer: order,
                    txID: txId
                };
                var transaction_id_string = txId.getTransactionID();
                var promises = [];
                var sendPromise = channel.sendTransaction(request);
                promises.push(sendPromise);
                peers.forEach(function(peer) {
                    let event_hub = channel.newChannelEventHub(peer);
                    let txPromise = new Promise((resolve, reject) => {
                        let handle = setTimeout(() => {
                            event_hub.unregisterTxEvent(transaction_id_string);
                            event_hub.disconnect();
                            resolve({ event_status: 'TIMEOUT' });
                        }, 40000);
                        event_hub.registerTxEvent(transaction_id_string, (tx, code) =>
{
                            clearTimeout(handle);
                            var return_status = { event_status: code, tx_id:
transaction_id_string };
                            if (code !== 'VALID') {
                                console.error('The transaction was invalid, code = ' +
code);
                                resolve(return_status);
                            } else {
                                console.log('The transaction has been committed on
peer ' + event_hub.getPeerAddr());
                                resolve(return_status);
                            }
                        }, (err) => {
                            reject(new Error('There was a problem with the
eventhub ::' + err));
                        },
                            { disconnect: true }
                        );
                        event_hub.connect();
                    });
                    promises.push(txPromise);
                })
                return Promise.all(promises);
            } else {
                console.error('Failed to send Proposal or receive valid response.
Response null or status is not 200. exiting...');
                throw new Error('Failed to send Proposal or receive valid response.
Response null or status is not 200. exiting...');
            }
        }).then((results) => {
            console.log('Send transaction promise and event listener promise have
completed');
            if (results && results[0] && results[0].status === 'SUCCESS') {
                console.log('Successfully sent transaction to the orderer.');
            } else {
```

```
                console.error('Failed to order the transaction. Error code: ' +
results[0].status);
            }
            if (results && results[1] && results[1].event_status === 'VALID') {
                console.log('Successfully committed the change to the ledger by the
peer');
                resolve('Successfully committed the change to the ledger by the
peer');
            } else {
                console.log('Transaction failed to be committed to the ledger due
to ::' + results[1].event_status);
            }
        }).catch((err) => {
            console.error('Failed to invoke successfully :: ' + err);
            reject(err);
        });
    })
}
```

Imagen 17: Código de la función instantiateChaincode

Esta función instancia el código del chaincode en el canal especificado.

La función hash no acepta ningún parámetro.

```
module.exports.hash = function() {
    return new Promise((resolve, reject) => {
        fs.readFile(path.join('..', '..', 'stonehenge.jpg'),
function(err, data) {
            let sha = crypto.createHash('sha256');
            sha.update(data);
            let hash = sha.digest('hex');
            resolve(hash);
        });
    })
}
```

Imagen 18: Código de la función hash

Esta función nos devuelve el hash 'SHA256' de la imagen.

La función putMetadata acepta un parámetro:

- ➔ channel: el nombre del canal que se quiere almacenar en los metadatos de la imagen.

```
module.exports.putMetadata = function(channel) {
    return new Promise((resolve, reject) => {
        cp.spawn('python', ['putMetadata.py', channel])
            .on("exit", () => resolve('Metadata updated'));
    })
}
```

Imagen 19: Código de la función putMetadata

Esta función ejecuta un script programado en Python llamado putMetadata.py que almacena el nombre del canal actual en la imagen en el campo UserComment definido en la especificación EXIF.

```
from PIL import Image
import piexif
import io
import codecs
import sys
channel = sys.argv[1]
op = io.BytesIO()
file = open("../../stonehenge.jpg", 'rb')
thumb_im = Image.open(file)
thumb_im.save(o, "jpeg")
thumbnail = op.getvalue()
exif_blockchain = {piexif.ExifIFD.UserComment: channel.encode('utf-8')}
exif_dict = {"Exif": exif_blockchain}
exif_bytes = piexif.dump(exif_dict)
piexif.insert(exif_bytes, "../../stonehenge.jpg")
```

Imagen 20: Script para insertar los metadatos

La función flatArray acepta un parámetro:

- ➔ array: la array que se quiere “aplanar”.

```
module.exports.flatArray = function(array){
    return array.reduce((acc, val) => Array.isArray(val) ?
acc.concat(this.flatArray(val)) : acc.concat(val), []);
}
```

Imagen 21: Código de la función flatArray

Esta función está definida recursivamente y nos permite normalizar los valores de las arrays anidadas dentro de la array especificada. El motivo de necesitar esta función se especificará más adelante.

4.5.3.4. Smart Contract

El chaincode requiere de la librería ‘fabric-shim’ proporcionado por Hyperledger Fabric para poder atender peticiones del peer para poder procesar transacciones o ejecutar consultas.

Además, la clase definida debe implementar dos funciones asíncronas: Init e Invoke.

La función Init se ejecuta cada vez que el chaincode se instancia o se actualiza mediante las peticiones correspondientes. En nuestro caso no haremos nada en ésta función.

La función Invoke es llamada cada vez que se realizan peticiones de tipo transaction o query. Es en esta función donde acabaremos ejecutando nuestras funciones.

Esta función tiene como parámetro de entrada un objeto del tipo ChaincodeStub, el cual es implementado por la librería 'fabric-shim' y pasado a las llamadas de Init e Invoke hechas por la plataforma Hyperledger Fabric. La clase ChaincodeStub encapsula las APIs entre la implementación del chaincode y el peer.

En la función invoke usamos la función de la clase ChaincodeStub llamada getFunctionAndParameters que nos devuelve un objeto que contiene el nombre de la función del chaincode llamada desde el servidor y sus parámetros. Según cual sea el nombre ejecutamos una función u otra.

Pasemos a explicar las funciones definidas en el Smart Contract.

Estas son:

- NewTransaction
- GetImageHistory

La función NewTransaction usa dos parámetros:

➔ stringTransaction: el objeto de la transacción en formato string.
➔ idImage: el nombre de la imagen.

```
async NewTransaction(stub, args) {
       console.info(args);
       let stringTransaction = args[0];
       let idImage = args[1];
       console.log(stringTransaction);
       try {
           return await stub.putState(idImage,
Buffer.from(stringTransaction));
       } catch (e) {
           return shim.error(e);
       }
   }
```

Imagen 22: Código de la función NewTransaction del Smart Contract

Esta función almacena en el canal la transacción en formato string y la asocia al nombre de la imagen, teniendo entre ellas una relación clave/valor.

La función GetImageHistory usa un parámetro:

➔ idImage: el nombre de la imagen.

```
async GetImageHistory(stub, args) {
        let idImage = args[0];
        try {
            let iterator = await stub.getHistoryForKey(idImage);
            let history = [];
            let done = false
            while (!done) {
                let item = await iterator.next();
                console.log(item)
                let buffer = item.value.value.toString('utf8');
                console.log(buffer)
                history.push(buffer);
                if (item.done == true)
                    done = true;
            }
            return history;
        } catch (e) {
            return shim.error(e);
        }
    }
```

Imagen 23: Código de la función GetImageHistory del Smart Contract

Esta función hace una consulta usando la función de la clase ChaincodeStub llamada getHistoryForKey, la cual nos devuelve el historial de todas las transacciones realizadas con el valor de idImage.

Es importante remarcar el caso en el que un canal tiene en su historial el historial del canal que ha heredado.

El historial del canal previo, que puede ser de varias transacciones, se almacenará como primera transacción del canal nuevo en forma de una array de transacciones en formato string. Si extrapolamos esta idea a la situación de varias herencias, obtenemos que cada historial heredado se ha ido anidando en varias arrays.

```
[
   [
      {
         "idImage": "stonehenge",
         "hashImage": "081fc7c055dad3ae8ac58619b9e9eff3b22d9d22dfbdb142b2097074c06d1daf",
         "newOwner": "user2",
         "license": {
            "adapt": false,
            "diminish": false,
            "embed": false,
            "enhance": false,
            "enlarge": false,
            "issue": false,
            "modify": false,
            "play": false,
            "print": false,
            "reduce": false
         }
      },
      {
         "idImage": "stonehenge",
         "hashImage": "081fc7c055dad3ae8ac58619b9e9eff3b22d9d22dfbdb142b2097074c06d1daf",
         "newOwner": "user3",
         "license": {
            "adapt": true,
            "diminish": false,
            "embed": false,
            "enhance": false,
            "enlarge": false,
            "issue": false,
            "modify": false,
            "play": false,
            "print": false,
            "reduce": false
         }
      }
   ],
   {
      "idImage": "stonehenge",
      "hashImage": "4c4236c2d75d8917866225677069 55e1ac8696dffb91346f033a97e1ee8556cb",
      "newOwner": "user1",
      "license": {
         "adapt": false,
         "diminish": true,
         "embed": false,
         "enhance": false,
         "enlarge": false,
         "issue": false,
         "modify": false,
         "play": false,
         "print": false,
         "reduce": false
      }
   }
]
```


Imagen 24: Ejemplo de un historial anidado

Justamente por este motivo se ha implementado la función flatArray mencionada anteriormente.

4.6. Test

Actualmente no existen herramientas estándar para poder realizar pruebas de test en Hyperledger Fabric, eso ha dificultado la realización de pruebas del software ya que por la condición de inmutabilidad de blockchain, cada vez que se hacían cambios en el código se debían volver a instanciar los contenedores para tener un entorno limpio.

Aún así este paso no conlleva mucho tiempo ya que docker facilita este tipo de acciones.

5. Resultados

En este apartado se explicarán las distintas aproximaciones que se han usado y el porqué se tomaron algunas decisiones en cuestión de la tecnología a usar y caso de uso a implementar.

Inicialmente se empezó con el desarrollo de una demo usando Ethereum con el caso de uso de registro de transacciones entre distribuidor y comprador. Se concluyó durante el desarrollo que una de las condiciones innatas de las redes públicas de una blockchain es que todos los participantes tienen una copia exacta de ella, por lo que resulta evidente ver que aunque favorece la seguridad de la red, también es un gasto muy elevado en términos de espacio de memoria de los participantes. Además, el hecho de que inevitablemente se necesite de un uso de Ether para realizar transacciones y que cada una de ellas debe ser ejecutada en todos los nodos, reduce significativamente la agilidad a la hora de registrar transacciones.

Finalmente se decidió dejar de lado esta tecnología e investigar la de Hyperledger Fabric, ya que tenía más sentido en un entorno B2B.

Hyperledger Fabric resultó una tecnología más compleja, ya que está ligado a más conceptos para poder implementar una solución.

Una vez familiarizados con Hyperledger Fabric, supuso mucho más natural el implementar un caso de uso orientado a B2B, ya que fue diseñada para ello. A eso también se debe añadir que resulta más ágil desarrollar y realizar transacciones con Hyperledger Fabric por el hecho de no depender de criptomonedas.

Aún así, se consideró que este caso de uso no cumple exactamente con la idea de generar una trazabilidad en la compra/venta de imágenes, ya que se centra en las transacciones entre distribuidor y comprador, y no en el historial de las imágenes en sí.

Finalmente, se decidió por Hyperledger Fabric aplicado al registro por imágenes.

6. Presupuesto

Se han dedicado aproximadamente unas 420 horas de desarrollo e investigación. A precio de ingeniero junior, aproximadamente 8€/hora, el coste seria de:

$$420\,horas \times 8\,\frac{€}{hora} = 3360\,€$$

El software utilizado es totalmente gratuito, así que no añade ningún tipo de coste adicional.

7. Conclusiones y líneas futuras

En este apartado se procede a valorar la aplicación de las tecnologías usadas en este proyecto en un escenario B2B.

Ethereum cumple con la visión clásica de una red blockchain totalmente descentralizada, pero el requerimiento de tener que usar criptomonedas para un uso en el que no interviene directamente el intercambio de dinero resulta una mala característica en este tipo de entornos.

Cierto es también que al no existir canales como en Hyperledger Fabric aporta una mejor robustez en cuanto a la seguridad de los datos registrados. Este último punto conlleva a un uso innecesario del almacenamiento de todos los nodos participantes de la red.

En cuanto a Hyperledger Fabric, es una tecnología más orientada al escenario propuesto, ya que es una red mucho más parametrizable a nivel de permisos y confidencialidad de la red, lo que deriva en un cumplimiento de los requisitos de las empresas interesadas.

Su desarrollo comparado con Ethereum es mucho más liviano una vez se ha familiarizado con sus particularidades.

Se concluye que en un escenario B2B, Hyperledger Fabric parece una tecnología más ideal, pero todo dependería de los requerimientos necesitados en cada caso.

Como posibles desarrollos futuros, se podría hacer uso de métodos más potentes para vincular los metadatos de la imagen con la blockchain, ya que en la demo implementada su uso es más bien anecdótico. Al tratarse de una demo, solo se ha trabajado con una imagen porque servía como prueba de concepto, pero la aplicación debería poder introducir cualquier imagen en el sistema.

Otro posible evolutivo sería combinar los dos casos de uso simultáneamente, ya que Hyperledger Fabric permite comunicación entre canales y en los peers se pueden instanciar más de un Smart Contract. Unos canales estarían destinados a registrar el historial de transacciones entre vendedor y comprador, y otros canales a registrar el historial de transacciones hechas sobre las imágenes.

Bibliografia

[1] "Ethereum Homestead Documentation". *Ethereum community*, 2016. [Online] Available: http://www.ethdocs.org/en/latest/index.html.

[2] "web3.js-Ethereum JavaScript API". *Ethereum*, 2019. [Online] Available: https://web3js.readthedocs.io/en/1.0/index.html.

[3] "Solidity". *Ethereum*, 2017. [Online] Available: https://solidity-es.readthedocs.io/es/latest/.

[4] "Hyperledger Fabric". *Hyperledger*, 2019. [Online] Available: https://hyperledger-fabric.readthedocs.io/en/latest/index.html.

[5] "Hyperledger Fabric SDK for node.js". *Hyperledger*, 2019. [Online] Available: https://fabric-sdk-node.github.io/release-1.4/index.html.

[6] "Hyperledger Fabric Node.js Contract and Shim". *Hyperledger*, 2019. [Online] Available: https://fabric-shim.github.io/release-1.4/index.html.

[7] "Angular". *Google*, 2019. [Online] Available: https://angular.io/docs.

[8] "Docker". *Docker Inc*, 2019. [Online] Available: https://docs.docker.com/.

Glosario

B2B: Business to Business

PoW: Proof of Work

MSP: Membership Service Provider

MVC: Modelo Vista Controlador

IDE: Integrated Development Environment

SPA: Single Page Application

SDK: Software Development Kit

EXIF: Exchangeable Image File Format