Departamento de (Ciencias da Computación e Tecnoloxías da Información)

Facultadde de Informática
UNIVERSIDADE DA CORUÑA

TRABAJO FIN DE GRADO
GRADO EN INGENIERÍA INFORMÁTICA
MENCIÓN EN TECNOLOGÍAS DE LA INFORMACIÓN

Herramienta de trazabilidad multidominio basada en Blockchain con Hyperledger Fabric

Estudiante: Miguel Gómez Rodríguez
Director/a/es/as: Antonino Santos del Riego

A Coruña, 15 de noviembre de 2019.

Dedicatoria

Agradecementos

Agradecer a mi familia y amigos por el apoyo prestado no solo en la realización de este trabajo, también a lo largo de estos cuatro años.

Resumen

Realizar el seguimiento de cualquier producto es una tarea que requiere gran cantidad de recursos y esfuerzo. La aparición de Blockchain ha supuesto un gran avance en este ámbito puesto que se adapta perfectamente a las características del dato, la traza. Además, si combinamos esta tecnología con Internet of Things(IoT) conseguimos reducir el error humano y automatizar al máximo el proceso.

El presente Trabajo Fin de Grado proporciona una herramienta para el diseño y despliegue de redes blockchain para el proceso de trazabilidad de cualquier tipo de producto independientemente de su domino de aplicación. Además, permite la interacción de dispositivos IoT con la red Blockchain para la consulta, modificación o incorporación de nueva información.

El uso de los frameworks de la tecnología Blockchain es ciertamente complejo. También se ha desarrollado una interface web para el framework Hyperledger Frabric que simplifica considerablemente la generación de las redes Blokchain y las automatización de sus procesos.

Con la finalidad de obtener un producto software de calidad, este trabajo ha seguido buenas prácticas de desarrollo y empleado una metodología ágil basada en incrementos. El resultado alcanzado es una herramienta con una arquitectura que permite la modificación y agregación de nuevas funcionalidades de forma sencilla.

Abstract

Tracking any product is a labor-intensive and resource-intensive task. The emergence of Blockchain has been a breakthrough in this area. since it adapts perfectly to the characteristics of the data, the trace. In addition, if we combine this technology with Internet of Things(IoT) we manage to reduce the human error and to automate to the the process.

This Final Degree Work provides a tool for the design and deployment of blockchain networks for the traceability process of any type of product regardless of its application domain. In addition, it allows the interaction of IoT devices with the Blockchain network for consultation, modification or incorporation of new information.

The use of Blockchain technology frameworks is certainly complex. A web interface has also been developed for the Hyperledger Frabric framework that considerably simplifies the generation of Blokchain networks and the automation of their processes.

In order to obtain a quality software product, this work has followed good development practices and employed an agile methodology based on increments. The result achieved is a tool with an architecture that allows the modification and aggregation of new functionalities in a simple way.

Palabras clave:

- Hyperledger
- Trazabilidad
- Blockchain
- Smart-contract
- Internet of Things

Keywords:

- Hyperledger
- Traceability
- Blockchain
- Smart-contract
- Internet of Things

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1
Introducción

Trazabilidad es la habilidad para trazar el histórico, la aplicación o la localización de un objeto según la ISO 9000 [1].

1.1 Motivación

Dentro del mundo globalizado en el que nos encontramos tener la certeza de que un objeto tiene un origen determinado o ha pasado unos controles específicos resulta una tarea sumamente importante. Por ejemplo, en la industria alimentaria, la Unión Europea establece una regulación que protege a los consumidores e impide la venta de alimentos no trazados [2]. Lo mismo ocurre con las industria farmacéutica, que está sometida a estrictos controles de calidad o con las denominaciones de origen, que necesitan auditorías periódicas.

Los problemas a la hora de realizar estos registros residen, por una parte, en la medición de las distintas dimensiones de la información del producto, que en una gran cantidad de casos se realiza a mano. Por otra parte, la recolección de esta información no suele estar centralizada, se dispersa a través del proceso de producción, lo que complica el acceso a toda la información de un producto[3].

Para solventar estos dos problemas la solución idónea es que la IoT y Blockchain trabajen conjuntamente. La primera recopila la información del producto y la segunda garantiza que esa información permanece inalterable, atribuible y consensuada por todos y cada uno los miembros de la red. De esta manera los datos recogidos son más fiables ya que no se someten a errores humanos y son accesibles por todos los componentes de la red Blockchain.

Tal y como se analiza en el apartado 2, existen varios tipos de redes blockchain, pero en este caso, debido a las propiedades del dato que vamos a manejar y los componentes que pueden formar parte de la red, ésta debe ser privada. Además, debemos poder autenticar a cada miembro de la red. Por último, la red debe emplear smart-contrats [4] para implementar la lógica en cada caso. Por estos motivos este trabajo utiliza el framework Hyperledger Fabric

que precisamente ayuda en la creación de una red que cumple todos estos requisitos.

Actualmente no existe ninguna herramienta en el mercado que genere soluciones que mediante la integración de ambas tecnologías. Los únicos proyectos existentes hasta la fecha se centran en desarrollos específicos en dominios de aplicación concretos, lo que supone una lenta implantación de la tecnología con un coste muy elevado.

1.2 Objetivos

Para resolver esta problemática este trabajo ofrece una herramienta capaz de definir cualquier tipo de red blockchain, pudiendo establecer el número de organizaciones, las conexiones entre estas o la cantidad de peers involucrados. Por la propia naturaleza del desarrollo, el framework Hyperledger Fabric se presenta como el idóneo para el planteamiento propuesto. Inicialmente se hace un completo análisis de este framework, de sus características y peculiaridades. Por distintos motivos, que se recogen el capítulo 5, el uso de dicho framework resulta bastante tedioso por lo que se decide desarrollar una librería que facilite su uso. Empleando esta última como base, se lleva a cabo el diseño e implementación de la herramienta.

Dentro del desarrollo de la librería se distinguen tres partes, un generador de certificados, el generador de archivos YAML con la descripción de la red y el generador de smart-contracts. La finalidad perseguida es la liberar al desarrollador del trabajo más manual, automatizando los procesos para que solo solo se necesaria la especificación de parámetros. Es nuestra librería la encargada de generar todo el material necesario para desplegar la red.

La implementación de la herramienta también se dividió en tres partes, por un lado, el desarrollo del modelo que proporcione todos los casos de uso referentes a la herramienta, por otro, la creación de una API REST que exponga todas esas funcionalidades y, por último, una interfaz web que interactúe con dicha API.

1.3 Breve descripción del trabajo

Este trabajo se divide en 8 capítulos.

El primero expone las carencias del proceso de traza actual y con qué tecnologías suplirlas. También define los objetivos alcanzados para el desarrollo la herramienta que integra ambas tecnologías.

El segundo capítulo introduce los conceptos menos conocidos pero fundamentales para el desarrollo del trabajo. Establece un punto de partida para la plena compresión del mismo.

El tercer capítulo agrupa las herramientas y tecnologías más importantes usadas durante la fase de desarrollo, así como los motivos de su elección.

El cuarto capítulo describe la metodología utilizada, Scrum, definiendo qué es un Equipo

Scrum, un Sprint, los artefactos, etc. Por otro lado también incluye el flujo de trabajo empleado, desde cómo se gestionaron los Sprints hasta cómo se trabajó con GitHub.

El quinto capítulo se centra en el análisis del framework Hyperledger Fabric, en qué consiste, cómo funciona y sus conceptos clave como pueden ser, peer, orderer, organización, canal o MSP. Por otro lado, también destaca los puntos débiles de dicho framework centrándose en el trabajo más tedioso que tiene que realizar el desarrollador y explicando posibles soluciones.

El sexto capítulo aborda el desarrollo de la librería que facilita el uso del framework y que formará parte de nuestra la herramienta. Automatizar la generación de certificados, smartcontracts o ficheros YAML con la descripción de la red son algunas de las tareas que desenvuelve la librería, permitiendo al desarrollador centrarse exclusivamente en declarar cómo quiere que sea la red.

El séptimo capítulo cubre el análisis, diseño, implementación y pruebas de la herramienta. Expone los casos de uso principales, el diseño de la interfaz, la creación de la API y la estructura del código. Además, este capítulo también recoge la definición e incorporación del smartcontract y la posterior interacción de dicho elemento con los dispositivos IoT.

El último capítulo aglutina las conclusiones, detalles a tener en cuenta sobre el trabajo realizado y futuros desarrollos que se pueden llevar a cabo sobre el software.

Capítulo 2

Conceptos previos

Actualmente la informática engloba una gran cantidad de áreas distintas, aportando cada una de ellas nuevos conceptos. Por desgracia, es muy complicado abarcar todos los conocimientos, por este motivo incluimos aquí los imprescindibles para la plena comprensión de este trabajo y que, por naturaleza, no son ampliamente conocidos.

2.1 Blockchain

Blockchain o cadena de bloques es una estructura de datos que almacena la información mediante bloques entrelazados de manera que su contenido solo puede ser editado o repudiado modificando todos los bloques anteriores. Esta propiedad permite su aplicación en un entorno distribuido que ejerce de base de datos no relacional y que contiene un histórico irrefutable de la información. Bajo un consenso, este entorno es capaz de alcanzar la integridad de los datos sin necesidad de recurrir a una entidad de confianza [5].

El contenido que se almacena puede ser muy diverso.Por ejemplo, si almacenamos transacciones o cambios de estado atómicos de tal manera que en su conjunto funcione como un libro contable obtenemos lo que se conoce como criptomoneda o criptodivisa.

La primera aplicación de esta tecnología fue Bitcoin [7], conocida por ser también la primera criptomoneda bajo el mismo nombre. Esta red almacena los movimientos de bitcoins entre carteras de forma descentralizada y sin ningún intermediario.

La cartera o *wallet* funciona como una cuenta corriente, tienen una dirección pública a la que se pueden asociar las criptomonedas. Gracias a la red blockchain se almacena el histórico de todas las transacciones, por lo que es sencillo saber la cantidad de criptodivisas de una cartera en un momento determinado. La gran popularidad de Bitcoin se debe a que establece un método de pago sin intermediarios ni comisiones [6][7].

2.1.1 Consenso

Es el mecanismo por el cual se añaden nuevos bloques a la red. Existen diferentes implementaciones pero todas se basan en lo mismo, se define el nuevo bloque, se genera y por último se valida por los demás miembros. Existen diferentes consensos en función de cómo se forma el bloque, entre los que destacan:

Proof of work. Este algoritmo se basa en resolver un problema de alta complejidad computacional que da como resultado el nuevo bloque. Este proceso se conoce como *mining* y a los nodos encargados de hacerlo son los *miners*. El problema está disponible para todos los *miners*, el primero que lo resuelva recibe una recompensa. Este tipo de consenso es el empleado en la red Bitcoin. Los inconveniente más destacados son, por un lado, que es necesario gran potencia computacional con su equivalente consumo de energía y, por otro, el tiempo de minado que, en el ejemplo mencionado, ronda los 10 minutos[8].

Proof of stake. Este algoritmo no permite que cualquiera genere el bloque, sino que es la propia red quien asigna a un nodo la tarea de crear el bloque. Esta asignación puede ser al azar, pero por lo general, se basa en factores como el *coin-age*, que es el resultado de multiplicar el número de criptomonedas por el número de días que las monedas han permanecido en esa cuenta.

2.1.2 Cadena

No existe blockchain sin una cadena, es una unidad fundamental. Está formada por bloques entrelazados, que almacenan una o varias transacciones, es decir, la cadena alberga todo el histórico de transacciones de la red. Como están entrelazados, en el momento que se añade un bloque a la red es imposible modificarlo sin modificar todos los anteriores, la cadena solo crece, por este motivo es inmutable. Por último, otro concepto importante y ampliamente utilizado es el *world state* o estado global, que simplemente refleja el estado tras realizar todas las transacciones de la cadena.[5][9]

2.1.3 Tipos de redes

Dependiendo de la fuente, el número de tipos de redes puede variar de 2 a 4. Siguiendo la tendencia actual, distinguiré solamente dos tipos de redes en función de quien puede formar parte de ella.

Redes privadas. Cada miembro de la red está identificado y es conocido, no todo el mundo puede formar parte de la red. Por lo general el proceso de consenso en esta red es mucho más rápido puesto que se confía en los miembros que la conforman. El framework que usaremos en este trabajo genera redes de este tipo.[10]

Redes públicas. Cualquiera puede ser miembro de la red, por lo general son anónimas y el mecanismo de consenso es mucho más complejo y costoso. Están formadas por muchos más nodos que las privadas lo que aumenta la disponibilidad de la red. Bitcoin [7] y Ethereum [6] son dos grandes ejemplos de redes públicas[10].

2.1.4 Smart-contract

Es un concepto que introdujo por primera vez Ethereum y que desde entonces ha sido ampliamente utilizado por la versatilidad que aporta a la redes blockchain. Los smart-contracts son programas informáticos que ejecutan acuerdos establecidos entre dos o más partes. Una de las partes clave es que al cumplirse las condiciones que se establecen en dicho programa, estos contratos inteligentes se ejecutan automáticamente, evitando la toma de decisiones esté centralizada.

Este avance se complementa muy bien con las redes blockchain puesto que nos proporcionan la inmutabilidad y el consenso entre todos los miembros, permitiendo crear sistemas completamente autogestionados y sin supervisión[11]. Los smart-contracts abren un gran abanico de posibilidades puesto que permiten establecer reglas complejas como, por ejemplo, la devolución del dinero en caso de retraso de un vuelo.

Token

Entre las posibilidades que ofrecen los smart-contracts destacan los tokens, que son unidades de valor emitidas por entidades privadas. El concepto es similar al de la criptomoneda pues cada token tiene valor y es intercambiable. La diferencia se encuentra en la forma, los tokens pueden ser capacidades de administración sobre un determinado sistema o los derechos sobre una parte de una vivienda. Esta forma se define en el smart-contract mediante el proceso que se conoce como *tokenización*. Esto abre un nuevo mundo de posibilidades, puesto que no solo se pueden intercambiar criptodivisas sino también cualquier otro token como inmuebles, derechos, capacidades, etc[12] [13].

2.2 Hyperledger

Es un proyecto de código abierto que tiene como objetivo hacer avanzar la tecnología Blockchain. Está amparado bajo la Linux Fundation y además cuenta con el apoyo de IBM, Intel o SAP Arriba entre otros. El software que engloba se divide en frameworks o herramientas.

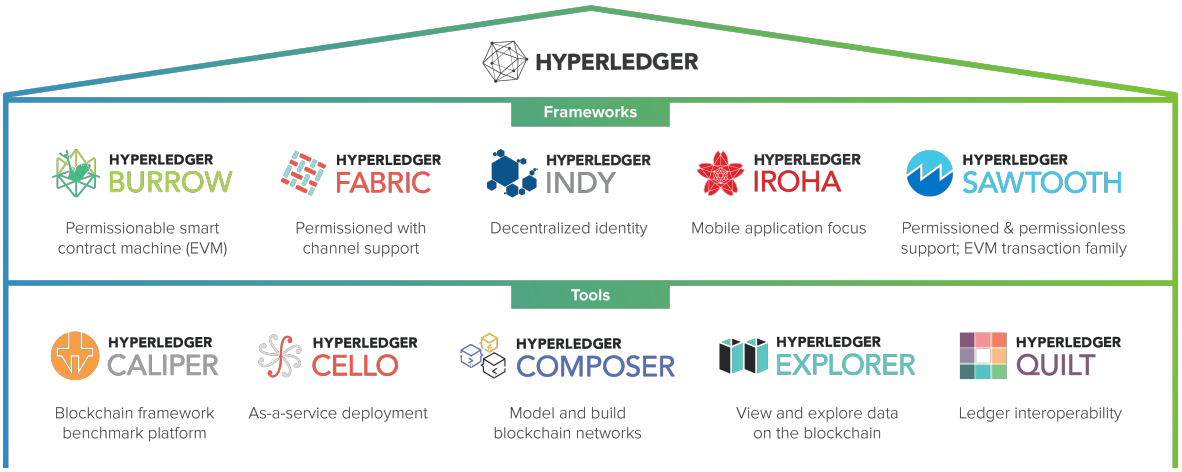

Figura 2.1: Diagrama del proyecto Hyperledger

El proyecto Hyperledger despliega diversos frameworks:

Fabric. Permite la generación de redes blockchain privadas para el uso de smart-contracts. Crea soluciones que tienen una arquitectura modular permitiendo que componentes como el consenso usado en la red o el servicio de acceso a la red, sean fácilmente intercambiables. Entre de todos los frameworks, este es el más avanzado, encontrándose actualmente en la versión 1.4 LTS.

Indy. Es una cadena distribuida para la creación identidades digitales. También engloba librerías y herramientas que permiten crear identidades digitales en diferentes blockchains o cadenas distribuidas y que estas sean interoperables entre dominios administrativos, aplicaciones y cualquier otro ámbito en las que se precisen [14].

Burrow. Es un cliente blockchain modular con un intérprete de smart-contract para la Ethereum Virtual Machine. El objetivo final de este proyecto es emplear los smart-contracts creados para redes Ethereum en una red creada con Fabric.

Entre las herramientas disponibles destacamos dos:

Explorer. Es una módulo blockchain que permite ver, solicitar o generar bloques, transacciones y datos asociados, información de la red, o los smart-contracts instalados por medio de una interfaz Web.

Composer. Es una herramienta que facilita la generación de redes con Fabric por medio de una interfaz Web. Pretende agilizar el proceso de creación de una red blockchain. Por desgracia, la red resultante permite pocas modificaciones, lo que ha motivado que no sea empleada en nuestro proyecto, hacia el desarrollo directamente sobre Fabric [15]

2.3 Docker

Es una plataforma de código abierto que permite la ejecución de aplicaciones sobre contenedores independientemente de la infraestructura que se encuentre por debajo. Un contenedor Docker es «an abstraction at the app layer that packages code and dependencies together» [16]. Estos contenedores son mucho más eficientes que las máquinas vituales puesto que solo necesitan las dependencias de la aplicación que se va a ejecutar. No requieren virtualizar la infraestructura ni instalar un sistema operativo sobre ella, sino que las aplicaciones se ejecutan sobre la capa docker que a su vez se encuentra por encima del sistema operativo anfitrión, tal y como se observa en la Figura 2.3

El uso de esta tecnología es importante para nuestro desarrollo porque desvincula la red blockchain que genera la herramienta del equipo en el que se quiera desplegar.

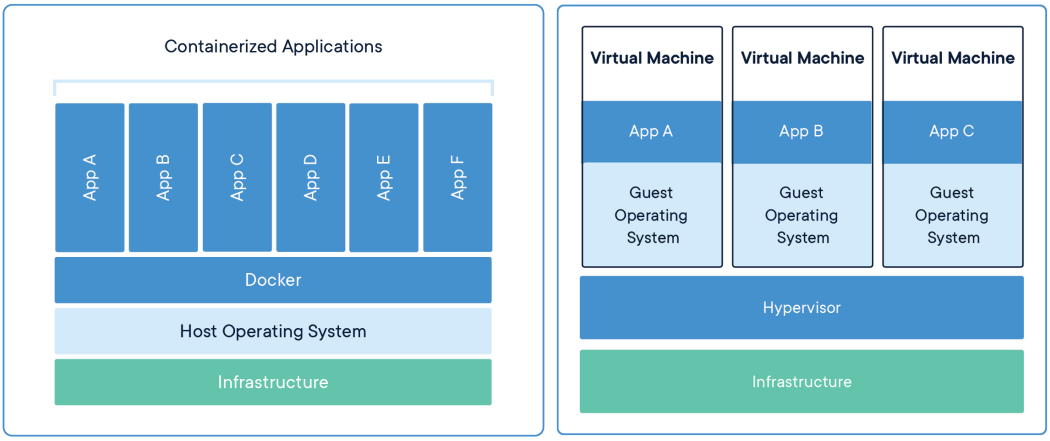

Figura 2.2: Esquema contenedor Docker frente a maquina virtual

La principal herramienta de esta plataforma es Docker Compose, que permite la creación de contenedores a partir de un archivo en formato YAML. En él se especifica la configuración referente a puertos, las imágenes que usa, si los contenedores pertenecen a una red o si cierto contenedor depende de otros. El framework que se utiliza en este trabajo emplea esta herramienta para la generación de la red.

Capítulo 3

Herramientas y tecnologías

Este capítulo recopila las tecnologías y herramientas que se han usado para el desarrollo de este trabajo y los motivos de su elección. Se ha dividido este capítulo en 4 partes bien diferenciadas, librería, backend, frontend y fase de desarrollo. Esta última agrupa las tecnologías empleadas durante todo el trabajo.

3.1 Librería

Es parte fundamental en el proceso de desarrollo, pues la herramienta delega en ella todo el cumplimiento de los requisitos. Las tecnologías empleadas han sido:

- NodeJS. Lenguaje de programación orientado a eventos asíncronos. El motivo de su elección fue, por un lado, porque es el mismo lenguaje que se emplea en el backend, y por el otro, porque el Hyperledger Fabric dispone de un SDK para Node.[17]

- Mustache. Es una librería capaz de autocompletar una plantilla por medio de un JSON con toda la información. El esquema que utiliza está formado por etiquetas entre llaves que se completan con los pares clave/valor del JSON. No solo busca la clave en la plantilla y la sustituye por el valor, también se pueden formar bucles que se ejecuten para todos los pares anidados dentro de otro. La figura 3.1 muestra un claro ejemplo [18].

3.2 Backend

Es la parte encargada de atender las peticiones HTML que realiza el frontend. Fue desarrollado en NodeJS puesto que es un lenguaje diseñado para el lado del servidor. Las tecnologías usadas durante su fase de implementación comprenden desde la ejecución de la aplicación, los tests y el procesamiento de peticiones:

View:

```
{
  "stooges": [
    { "name": "Moe" },
    { "name": "Larry" },
    { "name": "Curly" }
  ]
}
```

Template:

```
{{#stooges}}
<b>{{name}}</b>
{{/stooges}}
```

Output:

```
<b>Moe</b>
<b>Larry</b>
<b>Curly</b>
```

Figura 3.1: Ejemplo de uso de Mustache

- Express. Es una infraestructura de aplicaciones web Node.js mínima y flexible que proporciona un conjunto sólido de características para las aplicaciones web. Se emplea del lado del servidor, facilita la creación de endpoints y permite definir *middlewares* o funciones encargadas de atender las peticiones a ese endpoint. Además permite diferenciar entre métodos de petición HTTP y asignarlos a un *middleware* concreto [19].

```
var express = require('express')
var app = express()

var middleware = function(req, res){
    res.status(200).send()    //Envía una respuesta con código HTML 200
}

app.get('/route',middleware) //Todas las peticiones a /route serán atendidas por el middleware

app.listen(3000)
```

Figura 3.2: Ejemplo de uso de Express

- Chai. Es una librería para el desarrollo de guiado por pruebas, *Test Driven Development*, o por comportamiento, *Behavior Driven Development*, basada en aserciones. [20]
- Mocha. Framework de pruebas de JavaScript para Nodejs que ofrece la posibilidad de crear tanto tests síncronos como asíncronos [21].
- Nodedom. Herramienta que ayuda al desarrollo de aplicaciones Nodejs y que permite su ejecución, la detección de cambios en tiempo real y el reinicio de la misma [22].

3.3 Frontend

Entre todos los frameworks que existen en el mercado para crear entornos web se ha elegido Angular 7 en base a la experiencia disponible, ya se había trabajado con él en el pasado. La división del la interfaz en componentes reutilizables, el uso de servicios para acceso a la API y la interfaz de comandos facilitan el desarrollo de la vista de la herramienta. Las tecnologías utilizadas son:

- Angular 7. Framework de desarrollo de aplicaciones web basado en el Modelo Vista Controlador (MVC) desarrollado en Typescript. El modelo está formado por los servicios, los controladores son lo que se conocen como componentes y la vista está formada por las plantillas HTML. Además, también cuenta con una Command Line Interface (CLI) que facilita la creación de los elementos mencionados [23].
- Ngx-toastr. Paquete que incorpora un módulo para Angular que gestiona los *toasts* o notificaciones en la interfaz. Permite establecer distintos tipos de notificaciones, desde éxito, error o simplemente información. La herramienta la usa para informar al usuario de la finalización de una tarea en segundo plano y el estado de la misma [24].

3.4 Fase de desarrollo

Además de todas las tecnologías empleadas en cada una de las partes del trabajo, existen algunas que son usadas durante todo el desarrollo.

- NPM. Gestor de paquetes de Nodejs.
- Git. Software de control de versiones, almacena los distintos cambios que sufre el código fuente.
- GitHub. Repositorio online de proyectos que utilizan el control de versiones Git.
- Visual Studio Code. IDE de desarrollo de código abierto con una gran cantidad de extensiones.
- Draw.io. Conjunto de herramientas para la creación de diagramas online.

Capítulo 4

Metodologías

Para el desarrollo de la herramienta se ha seguido la metodología ágil Scrum, que divide el proceso de desarrollo en incrementos. El resultado de cada uno de ellos es un producto completamente funcional.

4.1 Scrum

Scrum es un marco de trabajo para el desarrollo y mantenimiento de productos complejos. Tiene como objetivo acometer problemas complejos adaptativos y a su vez entregar productos con el máximo valor tanto productivo como creativo. Está basado en la teoría de control de procesos de forma empírica, que gira entorno a tres pilares.

Transparencia. Todas las características clave del proyecto deben ser conocidas por todos sus miembros. Para ello el empleo de un estándar común aplicable a todos los proyectos facilita el entendimiento.

Inspección. Todos los miembros de un determinado proyecto deben inspeccionar los documentos generados por la metodología Scrum así como también los objetivos, para detectar variaciones que afecten tanto en tiempo como en forma y poder corregirlas.

Adaptación. Si un inspector determina una desviación fuera de los límites aceptables, el producto resultante no podrá ser aceptado. Se deberán tomar las decisiones necesarias para ajustar de nuevo el producto. Este ajuste se debe realizar lo más rápido posible para evitar desviaciones mayores.

El marco de trabajo está compuesto por Equipos Scrum, roles, eventos, artefactos y reglas asociadas.

4.1.1 Equipos Scrum

El equipo Scrum está formado a su vez por el Dueño del Producto (*Product Owner*), el Equipo de desarrollo (*Development Team*) y un Scrum Master. Cada equipo está formado por las personas necesarias para realizar el trabajo y no depende de nadie ajeno al equipo. Además, está dirigido por un miembro del mismo [25].

Dueño del Producto

Es el encargado de maximizar el valor del producto y del Equipo de desarrollo. También es el único encargado de gestionar la Lista del Producto (*Product Backlog*), se encarga de definir el orden de los elementos, de que el Equipo de Desarrollo conoce y comprende los objetivos definidos o de que cada elemento está redactado de forma clara para todo el Equipo [25].

Equipo de Desarrollo

El proceso de desarrollo se divide en incrementos. Es el Equipo de Desarrollo el encargado de realizar estos incrementos y entregar un producto terminado en cada uno de ellos. Estos equipos están autoorganizados, es decir, nadie externo toma decisiones para el equipo, y también son multifuncionales, cada equipo está formado por los miembros necesarios para realizar una tarea. El número de personas que forman un grupo es variable, aunque es recomendable que no exceda de 9 ni tampoco sea inferior de 3, pues requieren demasiada organización o bien los incrementos serían demasiado pequeños respectivamente [25].

Scrum Master

El Scrum Master es el encargado de asegurar que Scrum es adaptado y entendido por todo el equipo. También funciona como intermediario entre el exterior y el Equipo Scrum gestionando las interacciones.[25]

4.1.2 Eventos Scrum

Un evento Scrum es un bloque de tiempo que con una duración máxima y que tiene como finalidad regular y minimizar las reuniones. Existen 6 tipos de eventos Scrum [25].

El Sprint

El Sprint es concepto clave en el que se basa Scrum. Engloba todo el proceso de creación de un incremento hasta la entrega del mismo. Tiene una duración máxima de un mes debido a que más tiempo conllevaría una mayor complejidad y que a su vez aumenta los riesgos de que el incremento no se finalice.

Dentro de este evento, también se llevan a cabo otros, como son los Scrums Diarios, la Reunión de Planificación del Sprint, la Revisión del Sprint y la Retrospectiva del Sprint.

A su vez, dentro de cada Sprint también podemos encontrar distintas fases, se comienza con la definición de qué se va a construir, después se diseña y por último se implementa hasta tener el incremento terminado.

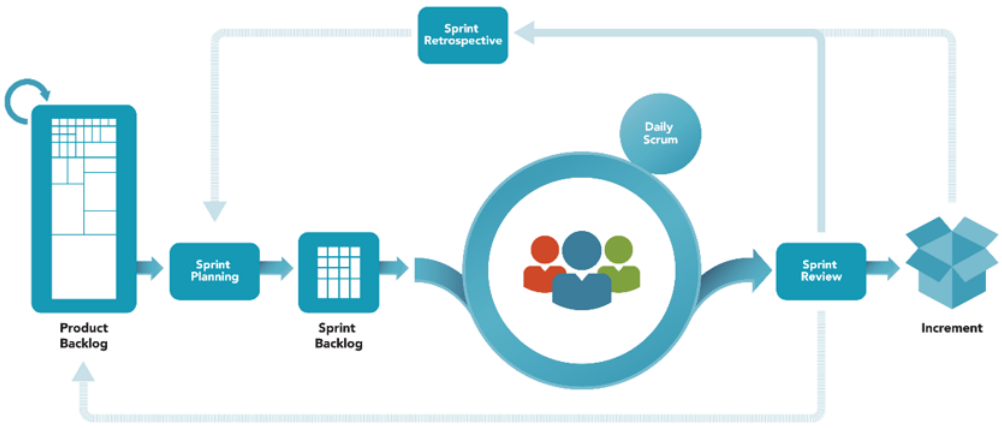

Figura 4.1: Mockup del segundo entorno de la interfaz

Reunión de Planificación del Sprint

Este evento se realiza solamente una vez durante todo el Sprint y determina, para el próximo Sprint, qué se va a desarrollar y cómo se va a desarrollar.

Objetivo del Sprint

Establece la meta que debe alcanzar el Sprint. Se marca durante la Reunión de Planificación en base a los elementos de la Lista de Producto. El Equipo de Desarrollo es el encargado de alcanzar ese objetivo, aunque este siempre ofrece cierta flexibilidad frente a la función implementada [25].

Scrum Diario

El Scrum Diario es una reunión del Equipo de Desarrollo de 15 minutos en las que se comprueba el progreso del incremento hacia el Objetivo Scrum y se marcan las actividades para las 24 horas siguientes. Este bloque de tiempo busca eliminar posteriores reuniones y agilizar el Sprint.

Revisión de Sprint

La Revisión del Sprint se lleva a cabo al final del mismo. Este bloque de tiempo tiene una duración máxima de 4 horas, en las que está presente el Equipo Scrum, el Dueño del Producto y sus invitados. Los asistentes colaboran para definir los siguientes pasos para optimizar el producto. Esta reunión no es de seguimiento sino que busca mejorar los incrementos posteriores poniendo en común los errores surgidos en el anterior y sus soluciones. Además, establece los elementos de la Lista de Producto posibles para el siguiente Sprint.

4.1.3 Artefactos Scrum

Los artefactos Scrum son representaciones del trabajo que se desarrolla durante el proceso Scrum que facilitan la transparencia de información acerca del proyecto a todo el Equipo Scrum.

Lista de Producto

La Lista de Producto agrupa de forma ordenada todos los requisitos que debe cumplir el producto. El Dueño de Producto es el único responsable del contenido, disponibilidad y orden de la lista. Cada elemento que compone establece características, funcionalidades, requisitos, mejoras y correcciones sobre el producto con el fin de ser entregadas en un futuro. La Lista de Producto nunca está terminada, a medida que el producto avanza también se detectan y añaden nuevos requisitos.

El refinamiento (*refinement*) consiste en añadir detalle, orden y estimaciones a cada elemento de la Lista de Producto. En este proceso colaboran tanto el Dueño de Producto como el Equipo de Desarrollo. Cuanto más alto en orden mayor es el detalle y viceversa.

Los elementos de la Lista de Producto que pueden ser terminados en un Spring se denominan ”preparados” o ”accionables” y ya son seleccionables por el Equipo de Desarrollo en una reunión de Planificación de Spring [25].

Lista de Pendientes del Spring

La Lista de Pendientes del Spring es el conjunto de elementos que pertenecen a la Lista de Producto que formarán el siguiente Spring. El Equipo de Desarrollo es el único encargado de mantener esta lista. A medida que el Spring avanza se eliminan elementos de la lista y, de la misma forma, cuando surge nuevo trabajo se añade a la lista. Por este motivo, este artefacto proporciona una visión en tiempo real de la línea de trabajo que seguirá el Equipo de Desarrollo para completar el proyecto [25].

4.2 WorkFlow

El desarrollo de este trabajo ha seguido la metodología Scrum con ayuda de GitHub para la creación y gestión de los incrementos. Debido a que el equipo está formado por una única persona, figuras como la del Scrum Master o la del Dueño del Producto se han visto reducidas. Sin embargo, eventos como la planificación del Sprint o el Scrum Diario no se han visto afectados.

Como veremos en los capítulos 7.3 y 6.3 se dividió en Sprints la implementación tanto de la libreía como de la propia herramienta. Siguiendo la planificación del Sprint, se definieron varios objetivos para cada incremento y que el producto debe cumplir. La duración de cada Sprint se estableció en dos semanas, aunque en algunos casos se desviaron a tres.

La gestión de los Sprint en este trabajo es a través de un proyecto de GitHub. Una vez definidos se establece una *milestone* o hito con el nombre del Sprint y una pequeña descripción. Después, para cada objetivo se crea un *issue* o tarea y se asocia a su hito, de esta manera a medida que se van completando las tareas el incremento va avanzando. A su vez cada *issue* creado pasa por tres estados antes de estar completado, como se puede ver en la Figura 4.2.

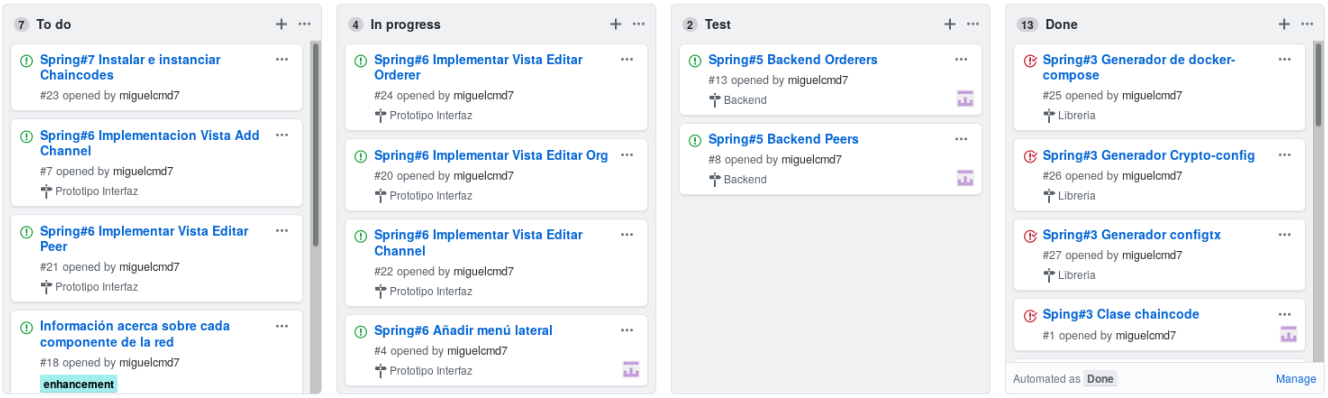

Figura 4.2: Gestión del trabajo

Por otro lado, el manejo de versiones se basó en dos ramas, por una parte la rama master, que almacena solamente las versiones terminadas del software, y por otro la rama dev, que recoge todo del desarrollo de la herramienta.

Capítulo 5

Análisis del framework Hyperledger Fabric

Para trabajar con este framework son necesarios una gran cantidad de conceptos nuevos que serían suficientes para un trabajo a parte. Por este motivo nos centraremos únicamente por sí solo en las partes más importantes y en el funcionamiento de la red con la finalidad de comprender el desarrollo de la herramienta.

5.1 Introducción

Una de las características clave es su modularidad y el alto grado de configuración que proporciona, pudiéndose modificar casi cualquier cosa. Este es uno de los motivos por lo que este framework está cobrando cada vez más fuerza. Permite establecer políticas complejas como que una transacción sólo se añada a la red si está firmada por ciertas identidades, establecer permisos de lectura, escritura o ambos sobre miembros concretos de la red, etc.

Está escrito en Go, por lo que la mayoría de novedades llegan antes a Go y luego a los distintos lenguajes compatibles por medio de los SDK's.

5.2 Conceptos clave

A la hora de trabajar con este framework es necesario conocer cómo funciona y los nuevos conceptos que introduce.

5.2.1 Peer

. Es la parte fundamental de la red además de la cadena. Toda blockchain está formada por *peers* que son los encargados de mantener la cadena y ejecutar los *smart-contracts*. Un peer puede albergar múltiples *smart-contracts* y múltiples cadenas. Cuando una aplicación quiere

consultar o añadir un bloque a la cadena debe conectar primero con el peer que almacena dicha cadena y además tener los permisos necesarios para acceder a ella, tal y como se muestra en la Figura 5.4 [26].

Un peer es la unidad básica, aunque puede tener variantes. Todos los *peers* de una misma organización se conocen entre si, pero cuando precisan comunicarse con un peer de otra organización necesitan disponer de ciertos detalles. Esta información se la proporciona lo que se conoce como *anchor peer* o peer ancla que debe formar parte de todas las organizaciones. Otra variante de peer, es el *endorser peer*, el que se encarga de firmar el resultado de las transacciones que quieren modificar la cadena. En el apartado 5.3 veremos en profundidad esta variante.

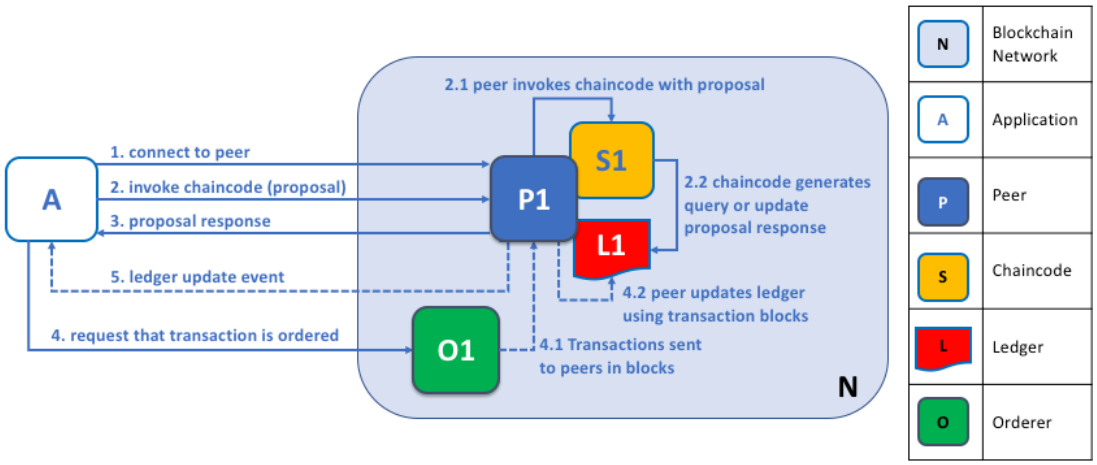

Figura 5.1: Esquema de conexión peer

5.2.2 Orderer

Cuando una transacción quiere actualizar el estado de la cadena es necesario generar un bloque con esa transacción que se añada a la cadena. Esta es precisamente la tarea que lleva a cabo el *orderer*, que por lo general no trabaja solo, formando otros *orderers* lo que se conoce como *orderer service*. Los detalles acerca de su configuración o funcionamiento como, las direcciones de los *orderes*, el tamaño máximo de bloque o de transacción entre otros, se almacena en el bloque génesis, que es el el primer bloque a partir del cual se forman las distintas cadenas de la red [27].

5.2.3 Channel

Si los *peers* son las unidades fundamentales de la red, lo que para Fabric se conoce como channel o canal son las estructuras que usan los *peers* para comunicarse entre si y mantener las cadenas. Es aquí también donde se instalan y distribuyen los *smart-contracts*. Los canales se pueden configurar estableciendo, por ejemplo, qué identidades pueden añadir más *peers* al canal o quienes pueden instalar nuevos *smart-contracts*. En el apartado 5.3 veremos cómo se realiza esta configuración [28].

5.2.4 Chaincode

Llegados a este punto, ya conocemos cómo funciona un *smart-contract* en una blockchain, aunque Fabric cambia este concepto, dado que entiende de forma diferente un *chaincode* de un smart-contract. La principal diferencia es que al primero se le pueden establecer políticas de acceso que no se establecen en la definición de *samrt-contract*. Aunque en la documentación sí se distingue, tanto para la comunidad como para este trabajo la nomenclatura no varía, se tratan como sinónimos. Actualmente la estructura de un *chaincode* se divide en dos partes, por un lado la función *Init*, que se ejecuta solamente una vez, y por otro la función *Invoke*, que por lo general redirige las llamadas a otras funciones, tal como se observa en la Figura 5.2 [11].

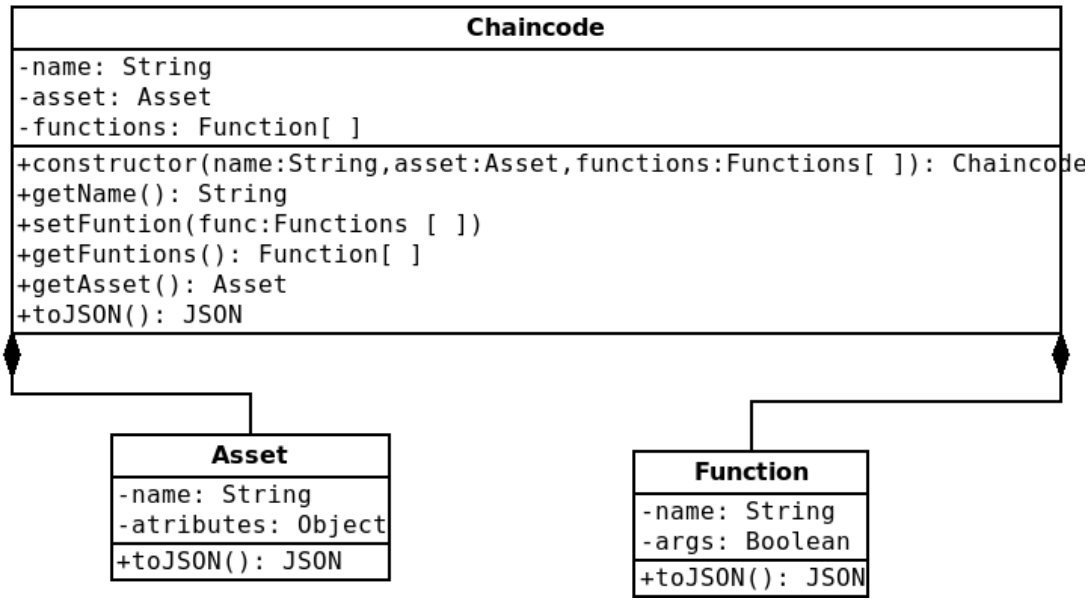

Figura 5.2: Esquema de *chaincode*

Para que desarrollo así como la red pueda emplear un *chaincode*, este debe instalarse en un

canal. Una vez instalado, para que se puedan ejecutar transacciones sobre él, se debe instanciar. Este proceso ocurre solo una vez y consiste en ejecutar la función *Init* que inicializa las variables necesarias. A continuación, ya se pueden llamar a las funciones que hemos definido por medio de un *invoke* y pasando como argumento el nombre de la función.

Las funciones que hemos definido pueden consultar el estado de la cadena o modificarlo, dependiendo de si es una u otra la interacción con el peer que alberga el *chaincode*. Si la transacción quiere modificar el estado de la cadena antes debe cumplir lo que se conoce como *endorsement policy*, que simplemente especifica qué organizaciones deben aprobar la transación para que la modificación se lleve acabo. Estas políticas se definen cuando se instala el *chaincode* en un canal.

Dentro del *smart-contract* se pueden realizar tres tipos de interacciones con la cadena:

- **Put.** Se emplea para crear nuevos objetos en la cadena.
- **Get.** Utilizada para recuperar objetos que ya están en la cadena.
- **Delete.** Borra los objetos del estado actual de la cadena, no del historial.

Por último, destacar que en Fabric se pueden implementar *chaincodes* en distintos lenguajes; Go, Nodejs o Java. Como el framework está escrito en Go, es este lenguaje el que incorpora las novedades más recientes. Este es el motivo por el que todos los *chaincodes* que se generan en este trabajo están escritos en este lenguaje.

5.2.5 Membership Service Provider (MSP)

Cada actor que pertenezca a la red como orderers, *peers*, administradores o usuarios necesita una identidad. Esta identidad se consigue mediante certificados digitales que proporciona una Autoridad Certificadora (AC). Por lo general, cada organización que forma parte de la red tiene su propia AC.

El MSP identifica a las AC's confiables para definir los miembros de un dominio, bien sea listando todos los miembros, identificando a la AC o, en la mayoría de casos, ambas. Además este servicio también permite definir los permisos que tienen cierta identidad o cierta AC sobre la red, como poder consultar, escribir bloques o ambos.

Tipos de MSP

Existen dos tipos de MSP dependiendo de dónde estén situados, el MSP local o el MSP de canal. El local se define para nodos de la red (*peers* u orderers) y en él se especifica, quien o quienes son los administradores, qué identidades tienen permiso para instalar un *smart-contract*, quien puede consultar la cadena, etc. Por otro lado, el MSP de canal establece la configuración del mismo, estableciendo quién puede añadir otros *peers* al canal o quién puede

instanciar *chaincodes*. Esta configuración se hace por medio de una transacción especial que se genera a partir del archivo configtx como veremos más adelante.

Estructura del MSP

Dentro del nodo se definen distintos apartados, los más destacados son:

Unidades Organizacionales (UO). No siempre es conveniente usar un único MSP para una organización. Es posible que esta se divida en departamentos y estos necesiten políticas distintas. Por esto nacen las UO, para dividir las unidades de una misma organización, como se observa en la Figura 5.3.

Administradores. Define las identidades que juegan el rol de administrador. Este no implica que pueda acceder a todos los recursos, sino que también está limitado a las políticas que se apliquen dentro de ese MSP.

Certificados Revocados. Al igual que se almacenan los certificados que son confiables también se guardan los certificados revocados para evitar su uso.

Identidad del Nodo. Como cualquier otro elemento de la red el MSP también tiene su propia identidad que se emplea para poder comunicarse con otros nodos de la red.

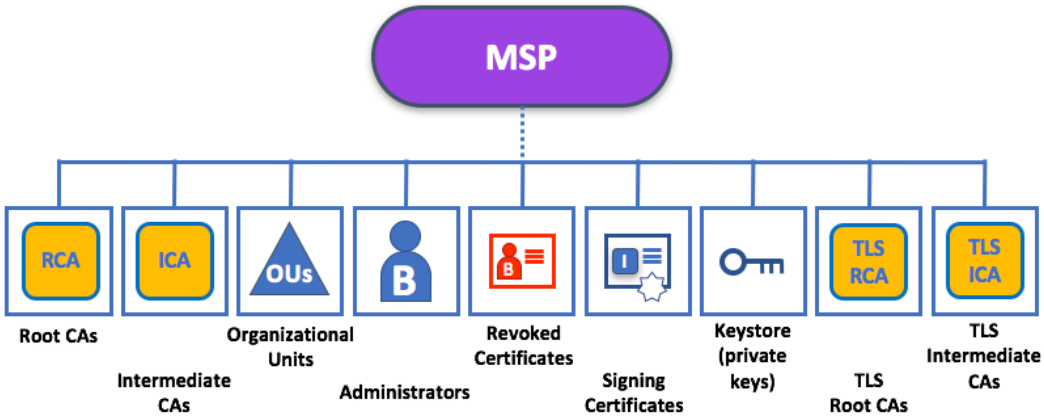

Figura 5.3: Esquema componentes MSP

Para concluir, el MSP funciona como un cortafuegos, define quién y cómo puede interactuar con qué. Es consultado cada vez que se quiere realizar una interacción con la red y este la autoriza o la deniega dependiendo de las políticas establecidas.

5.2.6 Organización

Este concepto es abstracto, pues no existe ninguna figura que represente una organización, como sí ocurre con los *peers* o los *orderers*. Se podría entender como organización a todos los *peers* que están bajo un mismo dominio, junto con sus MSP's y su AC.

5.3 Funcionamiento de la red

En este apartado se explica, por medio de los tres procesos más comunes en una red blockchain, como los conceptos analizados en el apartado 5.2 se relacionan entre si. Previo a los tres ejemplos suponemos que tenemos una red ya desplegada con uno *orderer*, varios *peers* y varias organizaciones.

5.3.1 Creación de canales

Aunque se despliegue una red, si los componentes no se pueden comunicar entre ellos no se podrían instalar *chaincodes* ni consultarlos, sería como no tener nada. Es por esto que la creación de canales es un punto muy importante dentro de Fabric.

1. El primer paso es conectarse a un *orderer* por medio de una aplicación. En el caso de que el usuario no sea aceptado por el MSP de nivel de red la conexión se rechaza.

2. A través de la conexión se realiza una petición de creación de canal especificando el nombre y la configuración del mismo, que se obtiene a partir del archivo configtx (ver apartado 5.4.1).

3. Si la petición se resuelve de forma exitosa, en el siguiente paso se obtiene el bloque génesis generado por el *orderer*. Por medio de una petición más sencilla que la anterior (no requiere parámetros) dicho nodo de la red nos devolverá la información.

4. En este punto el canal ya está creado, pero no tiene *peers* asociados. El proceso de añadir un *peer* consiste en enviar una petición de *join* en la que se adjunta el bloque génesis y el nombre del canal a cada *peer* que formará parte del canal. Dicho *peer* almacenará el bloque génesis y será este el primer bloque de todas las cadenas que formen los distintos *chaincodes* instalados en dicho canal.

5.3.2 Instalación e instanciación de chaincodes

Una red blockchain que no mantiene una cadena no aporta nada nuevo, es preciso instalar *smart-contracts* en los canales previamente creados para explotar todas las características que aporta Blockchain. Para poder realizar consultas sobre un *chaincode*, en Hyperledger Fabric

este debe estar instalado en un canal y posteriormente ser instanciado. En caso de simplemente estar instalado los *peers* solamente lo almacena, no lo ejecuta.

1. Por medio de una aplicación un usuario se conecta al *anchor peer* de su organización. El MSP de la organización puede aceptar o rechazar la conexión en función de los permisos que tenga dicho usuario.

2. Se le envía únicamente una propuesta de instalación al *peer* pasando como parámetro el nombre del *chaincode*, su *path*, el canal donde se instalará, la versión y el lenguaje.

3. En el caso de ser aceptado, el *anchor peer* distribuye el *chaincode* por todos los *peers* que pertenecen al canal y a su organización. La duración de este paso varía en función del número de nodos de la organización, pero una vez instalado podrá ser instanciado.

4. El siguiente paso consiste en establecer una conexión con cada uno de los *peers* que forman parte del canal y enviar una petición para instanciar el *chaincode* junto con la *endorsement policy* que veremos en profundidad en el apartado 5.3.3. En el momento que todos los *peers* respondan el *chaincode* pasará a ejecutar la función Init y permitirá las consultas.

Por lo general el usuario que realiza este proceso es un administrador que tiene acceso a todos los *peers* de la red, de esta forma no es necesario cambiar de usuario en función de la organización propietaria de los *peers*.

5.3.3 Consultar chaincode

Ahora que ya conocemos los componentes de la red, es necesario saber cómo interactuar con ellos. Una blockchain por si sola no nos aporta nada nuevo, es necesario que interaccione con el exterior. Este proceso ayudará a comprender la función de los componentes mencionados. Existen dos tipos de transacciones sobre los *chaincodes*, las que modifican la cadena y las que solo consultan. Siguiendo el esquema simplificado de la figura 5.4 se describe es este proceso:

1. Primero un usuario por medio de una aplicación se conecta con un *peer* de la red. El MSP a nivel de red identifica si este puede acceder a la red, en caso contrario la conexión se rechaza.

2. Se envía la transacción al *peer* con la función que se quiere ejecutar en el *chaincode* (suponemos que está ya instanciado). El *peer* ejecuta la función y es aquí donde el proceso difiere en caso de consultar la cadena o modificarla.

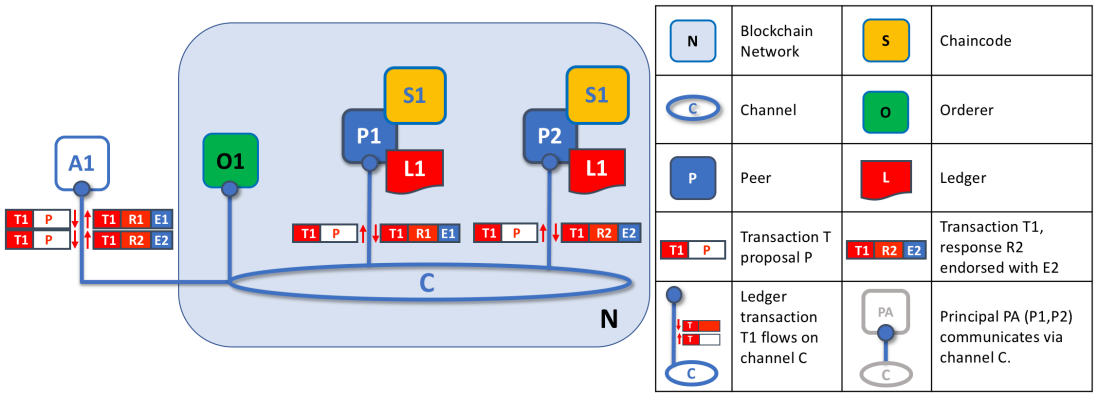

Figura 5.4: Interacción de una aplicación con la red blockchain

- Consulta: el *peer* consulta la cadena, devuelve el resultado y la transacción termina ahí.
- Actualización: en este caso, aparece la antes mencionada *endorsement policy* que establece qué organizaciones deben aprobar la transacción. Esta aprobación la realizan los *endorsement peers*. Cada organización posee uno o varios y son estos los que deben aprobar la transacción. El proceso de aprobación se produce ejecutando la función que modifica la cadena y firmando el resultado, esto es lo que se conoce como *endorser*. La aplicación deberá recibir todos los *endorsers* de los *endorsement peers* de las distintas organizaciones hasta que se satisfaga la *endorsement policy*. Durante este proceso ningún cambio se ha llevado a la cadena.

3. Una vez recibidos todos los *endorsers* la aplicación se los envía al *orderer service* formado por uno o más *orderers*. Es este el que se encarga de generar el bloque que será añadido a la cadena. Es el *orderer* quien se encarga de enviar el bloque a los *peers* para que actualicen su cadena y lo verifiquen.

5.4 Tipos de archivos

En este apartado se explica cómo se trabaja con el framework y las especificaciones que necesita.

5.4.1 Configtx

Este archivo aglutina distintas partes de la configuración global de la red. Por un lado, declara la configuración del *orderer service* definiendo valores como el tamaño de transacción,

la dirección de los distintos *orderers* o el tamaño de bloque entre otros. Por otro lado también especifica las organizaciones que forman parte de cada canal así como sus *anchor peers*.

```
Orderer: &OrdererDefaults

    # Orderer Type: The orderer implementation to start
    # Available types are "solo" and "kafka"
    OrdererType: solo

    Addresses:
        - orderer.miredseg.com:7051

    # Batch Timeout: The amount of time to wait before creating a batch
    BatchTimeout: 2s

    # Batch Size: Controls the number of messages batched into a block
    BatchSize:

        # Max Message Count: The maximum number of messages to permit in a batch
        MaxMessageCount: 10

        # Absolute Max Bytes: The absolute maximum number of bytes allowed for
        # the serialized messages in a batch.
        AbsoluteMaxBytes: 99 MB

        # Preferred Max Bytes: The preferred maximum number of bytes allowed for
        # the serialized messages in a batch. A message larger than the preferred
        # max bytes will result in a batch larger than preferred max bytes.
        PreferredMaxBytes: 512 KB
```

Figura 5.5: Archivo configtx

Gracias a la herramienta configtxgen podemos generar el bloque génesis con la configuración del *orderer service*, que se usará en cada cadena de la red. Con esta herramienta también podemos crear las transacciones de configuración para cada canal y la transacción que especifica qué *peers* funcionan como *anchors* para cada organización.

5.4.2 Crypto-config

Como explicamos en los apartados anteriores, cada elemento de la red tiene una identidad, este es el archivo que se emplea para generarlas. Se divide en dos partes dependiendo de lo que manejan las organizaciones:

PeerOrganizations. Son las organizaciones que disponen de *peers* en la red. Define el nombre de la organización, su dominio, el identificador de los *peers* y el número de usuarios que tendrá esa organización.

OrdererOrgs. En un primer momento esta sección servía para definir las organizaciones que controlaban los *orderers* aunque esta tendencia ha cambiado. Tal y como se especifica en

la documentación de Fabric el manejo de los *orderers* radica en una organización ficticia que se suele llamar *Orderer*. De esta manera evitamos el manejo de los *orderers* por un grupo distinto a las *PeerOrganizations*.

```
# ---------------------------------------------------------------------------
# "OrdererOrgs" - Definition of organizations managing orderer nodes
# ---------------------------------------------------------------------------
OrdererOrgs:
  # ---------------------------------------------------------------------------
  # Orderer
  # ---------------------------------------------------------------------------
  - Name: Orderer
    Domain: miredseg.com
    # ---------------------------------------------------------------------------
    # "Specs" - See PeerOrgs below for complete description
    # ---------------------------------------------------------------------------

    Specs:
      - Hostname: orderer
# ---------------------------------------------------------------------------
# "PeerOrgs" - Definition of organizations managing peer nodes
# ---------------------------------------------------------------------------
PeerOrgs:
  - Name: Org
    Domain: org0.miredseg.com
    EnableNodeOUs: true
    Specs:
      - Hostname: peer0 # implicitly "foo.org1.example.com"
      - Hostname: peer1 # implicitly "foo.org1.example.com"
    Users:
      Count: 1
```

Figura 5.6: Archivo crypto-config

Una vez definido el archivo, la herramienta cryptogen que proporciona el framework nos genera todos los certificados que necesita la red para ser desplegada en un carpeta denominada crypto-config.

5.4.3 Docker-compose

Este es el archivo más importante, especifica los contenedores que se van a crear, describiendo las rutas de los certificados que necesita, los puertos, la dirección del nodo, el nombre del MSP así como otras variables. Aquí se define la red, cuantos *orderers*, *peers* y AC's se necesitan. Este archivo necesita conocer dónde se almacenan los certificados y el bloque génesis de la cadena. Por este motivo, que debe crearse después de los dos archivos definidos anteriormente.

Gracias a la herramienta Docker Composer es posible ejecutar la misma red en distintas infraestructuras simplemente con disponer de este archivo.

5.5 SDK's

Siglas de *Software Development Kit*, consiste en una colección de herramientas software para el desarrollo de aplicaciones. Por lo general esta colección está desarrollada en distintos lenguajes para facilitar adaptarse a distintos desarrolladores. Dentro de Fabric podemos distinguir dos tipos de SDK's:

Smart-contracts. Desde Hyperledger se está trabajando para permitir su desarrollo en distintos lenguajes, entre ellos, Solidity, el lenguaje empleado en Ethereum, con el fin de atraer a los desarrolladores de esta red.

Interación con la red. También disponible en varios lenguajes, Nodejs, Java y Python, aunque este último está en fase inicial. Facilita la comunicación con la red, permitiendo la instalación de *chaincodes*, su ejecución o la creación de canales entre otras.

5.6 Carencias destacables

Los puntos débiles de este framework se centran en la usabilidad del mismo. Para crear una simple red se necesita crear como mínimo tres archivos. Se pueden usar los ejemplos, aunque son poco adaptables a modelos concretos. Todo ello produce los siguientes problemas:

Modificación manual de archivos. Para variar cualquier característica de los ejemplos o bien crear una nueva red, se deben definir los archivos manualmente, lo que es un proceso engorro. Por ejemplo, cuando se generan los certificados, el nombre de fichero con la clave privada cambia y es necesario volver a modificar el docker-compose.yaml.

Creación de canales. Encontramos dos variantes para este proceso:

- Conectándose directamente al *peer*, ejecutar los comnados necesarios y unir a los distintos *peers*.
- Mediante el uso un SDK. El proceso es más sencillo ya que existen funciones de instalación, pero cada vez que quieras modificar el canal o formar uno nuevo, se deberá escribir código.

Por otro lado, es necesario adquirir nuevos conocimientos que son solo aplicables a esta red, es decir, si vienes de Ethereum, conociendo los *smart-contracts* y las redes blockchain adaptarte a Fabric será complicado.

Capítulo 6

Desarrollo de la librería de interacción con el framework Hyperledger Fabric

Antes de comenzar con el desarrollo de la herramienta fue preciso especificar a alto nivel cómo sería el software resultante. Una vez estudiado el framework Hyperledger Fabric, se encontraron dos clases de requisitos claramente diferenciados. Por una lado, los referentes a la interacción con dicho framework, tanto para el despliegue como para el diseño de la red y, por otro lado, los relativos a la interfaz y la interacción del usuario final con la herramienta. Siguiendo esta diferenciación para este trabajo se realizaron dos desarrollos, la librería y la herramienta.

Este capítulo recoge las distintas fases del desarrollo e implementación de la librería pues es la que proporciona el soporte a la herramienta. Como se especificó en el capítulo 4 este trabajo emplea la metodología Scrum, por lo que la primera fase se centra la definición de requisitos en el *Product Owner*. La siguiente consistió en dividir los elementos de dicha lista en Spring para su diseño e implementación. Por último se hacen las unas pruebas de unidad y rendimiento.

6.1 Objetivos

La finalidad de la librería es, por un lado, facilitar el uso del framework resolviendo la carencias vistas en 5.6 y, por otro lado, proporcionar un soporte para el desarrollo de la herramienta. El primero objetivo se resolvió por medio de la automatización de la mayoría de procesos del framework y el segundo proporcionando una capa de interacción uniforme con la librería.

6.2 Análisis de requisitos

Es la primera fase del desarrollo de software y en ella se definen las especificaciones que debe cumplir el software resultante. Es una parte fundamental porque muestra una visión global de todo el trabajo y ayuda a anticipar posibles inconvenientes a la hora de diseñar.

6.2.1 Requisitos funcionales

Las características que debe cumplir el software una vez detectados los requisitos son:

- Generar archivos YAML
  - Generar docker-compose
  - Generar configtx
  - Generar crypto-config
  - Ejecutar configtxgen
  - Ejecutar cryptogen
- Definición de la red
  - Creación, modificación y eliminación de *peers*
  - Creación, modificación y eliminación de orderes
  - Creación, modificación y eliminación de organizaciones
  - Creación, modificación y eliminación de canales
- Despliegue de la red
- Instalación de de chaindodes
- Generar chaincodes en GO
- Generar plantillas de interacción con la red para dispositivo IoT en Python

6.2.2 Requisitos no funcionales

Una vez definidos los requisitos que debe cumplir, también se deben tener en cuenta otras características del software resultante.

- Multiplataforma. Debido a que esta librería será usada por la herramienta, al igual que ella debe poder ejecutarse sobre cualquier equipo.

6.3 Diseño y planificación

El diseño es una parte crucial en el desarrollo software pues ayuda a crear una imagen global del producto y anticipa los problemas que puedan surgir en la fase de implementación. Este apartado recoge el diseño general de la librería así como su división en Sprints. Además, para cada uno de ellos también se recoge el diseño específico de ese incremento y sus objetivos. En el apartado 6.4 se incluyen los aspectos más destacados de la fase de implementación de la librería.

6.3.1 Diseño general

Antes de dividir el desarrollo en incrementos es preciso tener una visión global de cómo va a ser el software resultante. En este caso, para cumplir los requisitos, se ha dividido en tres módulos:

Modelos Son las clases que conforman la estructura básica de la red blockchain. *Network*, *Peer*, *Orderer* y *Organization* pertenecen a este grupo y siguen el diagrama UML de la Figura 6.1

Funciones de utilidad No pertenecen a ninguna clase en concreto, son funciones auxiliares que ayudan a cumplir los requisitos funcionales. Aquí se agrupan los generadores, las funciones encargadas de desplegar la red o las de instalar los *chaincodes*.

Módulos de interacción Cada uno de ellos recoge las funciones que expone la librería para un componente de la red específico.

6.3.2 Sprint 1

El primer incremento debe formar la estructura básica de una red, por este motivo se diseñan e implementan las clases *Peer*, *Orderer* y *Organization*. Una vez analizado el framework Hyperledger Fabric simplemente añadimos los atributos que definen cada clase. Por ejemplo, en el caso de *Peer*, es necesario el id, el dominio de la organización a la que pertenece, los puertos para comunicarse con él y si es *anchor* o no. Para estos dos últimos datos se decidió crear una clase aparte, PeerConf, tal y como se observa en la Figura 6.1.

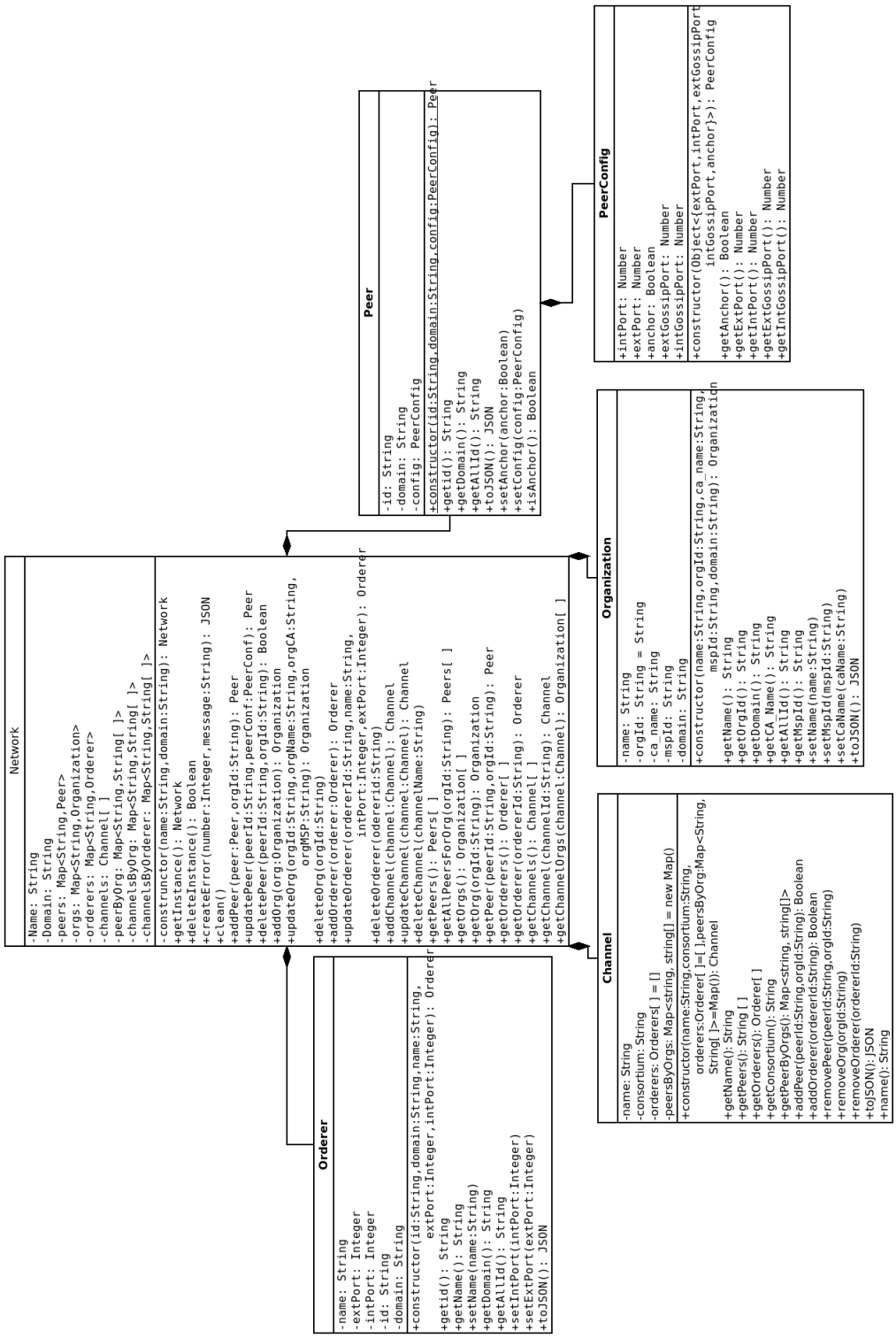

Figura 6.1: Diagrama UML de las clases modelo de la librería

Objetivos

Los objetivos para este Sprint fueron:

- Definición de los atributos de identificador, dominio, puertos internos y externos para la clase *Peer*, así como sus respectivos *getters* y *setters*.
- Definición de los atributos de nombre, identificador, dominio, puertos internos y externos para la clase *Orderer* y así como sus respectivos *getters* y *setters*.
- Definición de los atributos de identificador, dominio, MSP y AC para la clase *Organization* y así como sus respectivos *getters* y *setters*.

6.3.3 Sprint 2

Con la definición de la estructura básica del sprint anterior, el siguiente paso es la creación de la clase *Network* siguiendo la Figura 6.1. Los atributos que pertenecen a esta clase son: el nombre, el dominio y listas con los *peers*, los *orderers* y las organizaciones que componen la red. Por otro lado, para esta clase se ha seguido el patrón *Singleton*.

Singleton

Singleton o instancia única es un patrón de diseño que restringe la creación de objetos de una clase, solo permite uno. El motivo para usar este patrón es evitar el excesivo uso de instancias iguales de un mismo objeto. Además como veremos son instancias muy costosas en cuanto a memoria.[29]

Objetivos

Los objetivos para este Sprint fueron:

- Definición de los atributos nombre y domino de la clase *Network* así como sus getters y setters.
- Métodos para añadir, actualizar y eliminar *peers*.
- Métodos para añadir, actualizar y eliminar *orderers*.
- Métodos para añadir, actualizar y eliminar organizaciones.

6.3.4 Sprint 3

Una que se tiene el esquema de a red, el siguiente paso se centra en generar los archivos y material criptodgráfico necesarios para lanzar la red. Este proceso se consiguió por medio de plantillas que se completan con los pares clave/valor de los archivos JSON gracias a Mustache. Lo primero que se diseñó fueron las plantilas y una vez terminadas se decidió añadir métodos a las clases creadas en los dos incrementos anteriores para devolver los pares clave/valor de cada objeto. Por último se diseñaron las funciones *configtxYaml*, *crytoYaml* y *dockerYaml* que combinan las plantillas con el resultado de los métodos anteriores.

Objetivos

Los objetivos de este incremento fueron:

- Generar el archivo docker-compose
- Generar el archivo configtx
- Generar el archivo crypto-config
- Desplegar la red.

6.3.5 Sprint 4

Cuando la red está desplegada en el siguiente paso se deben crear los canales. Inicialmente creamos el modelo mediante una clase que define qué es un canal y añadí un atributo tipo lista a la clase Network. En siguiente paso se diseña el módulo de utilidad para los canales que, a su vez está compuesto un submódulo, ChannelCreator, que emplea el SDK de Fabric para interactuar con la red, crear canales y unir a los *peers* necesarios. Una vez terminado también se diseña el *ModuloChannel* que expone las funciones de la librería referentes a los canales.

Objetivos

Los objetivos de ente Sprint fueron:

- Definición de los atributos consorcio, lista de *peers*, lista de *orderers* y consorcio de la clase *Channel*.
- Métodos para añadir y eliminar *peers* del canal.
- Métodos para añadir y eliminar *orderers* del canal.
- Crear y añadir los *peers* en al red desplegada.

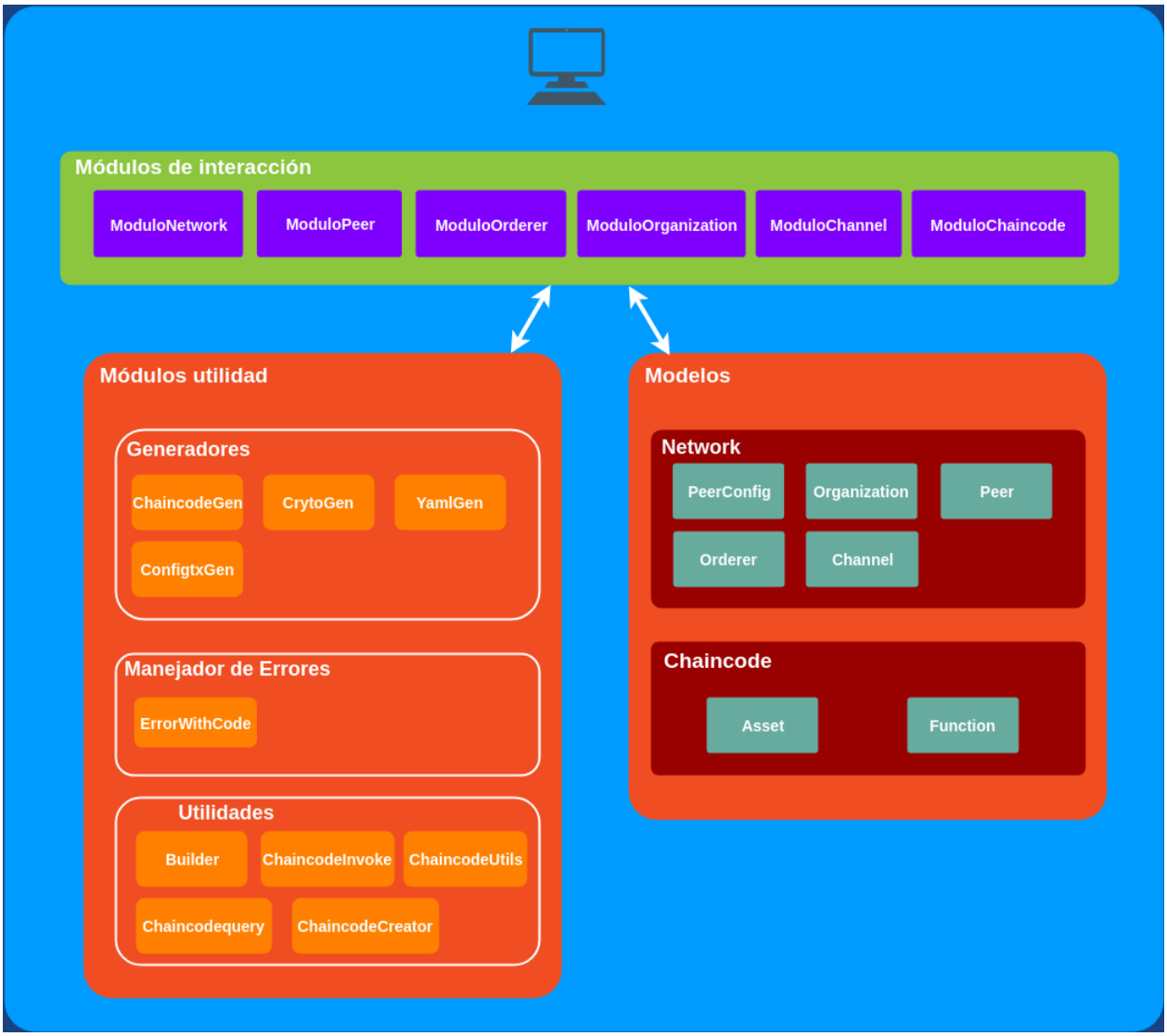

Figura 6.2: Módulos de la librería

6.3.6 Sprint 7

El último incremento define qué es un *chaincode*, mediante la creación de una nueva clase. Como nuestra herramienta tiene la finalidad de trazar cualquier objeto, se definen dos componentes que deberían tener todos los *chaincodes*. El *asset* u objeto a trazar y las funciones que expone el *chaincode*. Para estos dos componentes se crearon tres clases siguiendo el UML de la Figura 6.3. Una vez definido el *chaincode*, el siguiente paso se generan las plantillas, para lo que hizo falta un generador y una plantilla de Mustache.

Por otro lado, este Sprint también contempló el desarrollo de las plantillas para dispositivos IoT, para lo que fue necesario de nuevo, una plantilla Mustache junto con su correspon-

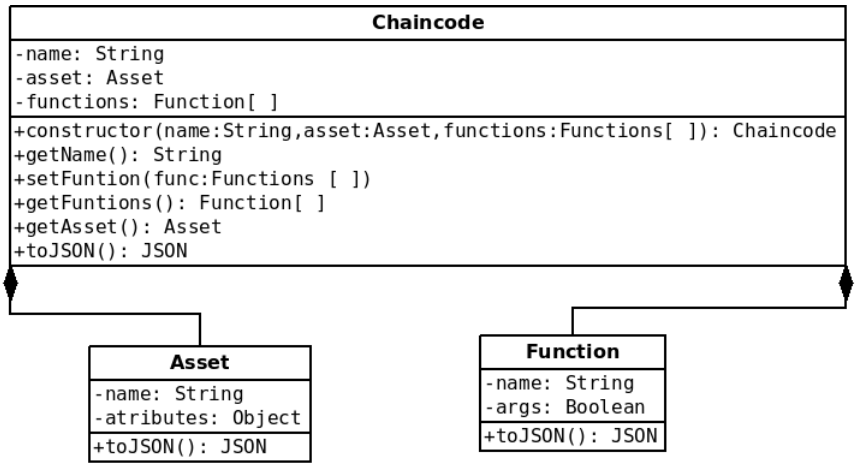

Figura 6.3: Diagrama de UML *chaincode*

diente generador.

Objetivos

Los objetivos de ente Sprint fueron:

- Definición de *Asset*, *Function* y *Chaincode*
- Generador de *chaincodes*
- Generador de plantillas para dispositivo IoT
- Crear y añadir los *peers* en al red desplegada.

6.4 Implementación

Aunque esta fase se realizó para cada uno de los Sprints, este apartado recoge los aspectos más destacados de librería de forma global.

6.4.1 Generadores

Una parte clave para aumentar la usabilidad del framework fue la generación de archivos de forma automática. Esta tarea se divide en cinco en función de los distintos tipos de archivos que genera la herramienta. Destacan por una parte los generadores para los archivos propios de Fabric, que son CryptoGen, ConfigtxGen y Yamlgenerator. Por otra parte el generador de *chaincodes*, que permite la definición de un *asset* y sus funciones él en el lenguaje de programación Go. Por último, el generador que permite la interacción de dispositivos IoT con la red blockchain por medio de Python.

Todos los generadores se basan en lo mismo, una plantilla de Mustache que se combina con un JSON para completarla. Esta plantilla varia dependiendo del archivo.Por este motivo la clase *Network* tiene tres métodos distintos para convertir a JSON, uno para cada archivo.

```
# ---------------------------------------------------------------------------
# "OrdererOrgs" - Definition of organizations managing orderer nodes
# ---------------------------------------------------------------------------
OrdererOrgs:
  # ---------------------------------------------------------------------------
  # Orderer
  # ---------------------------------------------------------------------------
  - Name: Orderer
    Domain: {{netDomain}}
    # ---------------------------------------------------------------------------
    # "Specs" - See PeerOrgs below for complete description
    # ---------------------------------------------------------------------------

    Specs:
    {{#orderers}}
      - Hostname: {{ordererId}}
    {{/orderers}}
# ---------------------------------------------------------------------------
# "PeerOrgs" - Definition of organizations managing peer nodes
# ---------------------------------------------------------------------------
PeerOrgs:
  {{#peerByOrgs}}
  - Name: {{orgName}}
    Domain: {{domain}}
    EnableNodeOUs: true
    Specs:
    {{#peers}}
      - Hostname: {{PeerId}} # implicitly "foo.org1.example.com"
    {{/peers}}
    Users:
      Count: 1
  {{/peerByOrgs}}
```

Figura 6.4: Plantilla Mustache de crypto-config

6.4.2 Errores

Por defecto, las excepciones que lanza Nodejs no aportan mucha información acerca de porqué se han producido. Es por este motivo que implementé la clase ErrorWithCod que hereda de la clase Error de Nodejs y además le añade los atributos *code* y *error_message* que facilitan la compresión del motivo del error.

6.4.3 Util

Por medio de los modelos definimos cómo queremos una red, aunque se queda en una mera especificación. Gracias a este módulo Dentro de los objetivos de la librería se encuentran el despliegue de la red y la instalación e invocación de chaincodes. Estas funcionalidades están implementadas en cuatro clases

Builder. Esta clase agrupa las llamadas a los generadores y los métodos para el despligue de la red. Por una lado nos encontramos con el método *build*, que recibe por parámetro

Figura 6.5: Ejemplo JSON para *parsear* configtx

Figura 6.6: Implementación ErrorWithCode

la configuración de una red de la que exporta su configuración a los ficheros configtx, crypto-config y docker-compose.yaml. Por otro lado, el método *launch* emplea estos archivos para desplegar la red. Como no existen SDK's para realizar esta función, dicho método emplea la librería *child_process* que, entre muchas funciones, permite la ejecución de los comandos que se le pasen por parámetro en un proceso a parte. En la Figura 6.7 se observa cómo por medio de esta librería se automatiza el despliegue de la red.

```
async function launch(net){
    return new Promise((resolve,reject)=>{
        stopNetwork()
        let networkYaml = gen(net)
        fs.writeFileSync(_(['docker-compose.yaml']), networkYaml);
        //execSync('configtxgen -configPath '+ process.env.DEST_DIRECTORY+' -profile One
        execSync('docker-compose -f ' + _(['docker-compose.yaml']) + ' up -d ');
        setTimeout(async ()=>{
            for (let channel of net.getChannels()){
               await  ChannelBuilder.initChannel(channel,net);
            }
            resolve()
        },1500)
    })
}
```

Figura 6.7: Uso de la librería *child_process*

ChannelCreator. Esta clase es la encargada de crear los canales una vez inicializada la red. Únicamente recibiendo como parámetros el canal y la configuración de la red, el método *initChannel* es capaz de obtener el *orderer*, el bloque génesis y añadir a todos los *peers* de dicho canal. Para ello emplea el SDK de Fabric que simplifica la comunicación con la red.

ChaincodeInstaller. Por medio de esta clase se realiza la instalación de los smart-contract en la red. Al igual que la clase anterior, también se emplea el SDK de Fabric. Se recibe como parámetros la configuración de la red, el chaincode y el canal donde se va a instalar el método *installChaincode*. La clase elige el *anchor peer* para enviar la petición de instalación, una vez aceptada envía una petición de instanciación a todos los *peers* del canal junto con la *endorsement policy*, tal y como podemos ver en la Figura 6.8.

```
const request = {
    targets:targets,
  chaincodePath: chaincode_path,
  chaincodeId: chaincode_id,
  chaincodeVersion: version,
  fcn: 'init',
  args: [],
  txId: tx_id,
  chaincodeType: type,
    // use this to demonstrate the following policy:
    // 'if signed by org1 admin, then that's the only signature required,
    // but if that signature is missing, then the policy can also be fulfilled
    // when members (non-admin) from both orgs signed'
  'endorsement-policy': {
    identities: [
      {role: {name: 'member', mspId: mspId}},
      {role: {name: 'admin', mspId: mspId}}
    ],
    policy: {
      '1-of': [
        {'signed-by': 1},
        {'signed-by': 0}
      ]
    }
  }
};
```

Figura 6.8: Ejemplo petición de instanciación

ChaincodeQuery. A través de ella se gestionan todas las consultas a los *chaincodes*. Simplifica notablemente esta tarea pues escoge ella sola la identidad correcta para realizar

la petición. De esta manera el desarrollador solo se preocupa del método al que quiere llamar y la librería se encarga de devolver el resultado de la consulta.

```
await getAdmin(client,network.getOrg(orgId))

const request = {
    chaincodeId : chaincodeId,
    fcn: fcn,
    args: args,
    request_timeout: 3000
};


console.log("Query chaincode");
let response =  await channelReq.queryByChaincode(request);
console.log("Query response:");
console.log(response[0]);
```

Figura 6.9: Ejemplo query chaincode

ChaincodeInvoke Esta clase realiza una función similar a la anterior con la salvedad de que para hacer una petición de tipo Invoke se debe satisfacer la *endorsement policy*. Por este motivo, la primera tarea que se realiza es obtener los *endorsers* de los *endorsement peers* y después enviar esas respuestas a *orderer service* para generar un bloque que se enviará a los *peers*. Al igual que en la clase *ChaincodeQuery* el proceso es totalmente transparente para el desarrollador y los parámetros de entrada no difieren.

6.4.4 Índices

Bajo las pruebas sobre la librería se observó que el rendimiento empeoraba a la hora de devolver los *peers* de una organización y los canales a los que pertenece la organización. Por esta razón, se implementó unos índices calve/valor para ambos casos. En el primero asocié el identificador de la organización con una lista de identificadores de *peers*. En el segundo caso también se asoció el id de la organización, pero esta vez con una lista con los nombres de los canales. De esta manera la búsqueda es mucho más eficiente solo sobre texto y no sobre listas de objetos.

```
/**
 * @type {Map<string,string[]>}
 */
this.peerByOrgs = new Map();
/**
 * @type {Map<string,string[]>}
 */
this.channelsByOrg = new Map();
```

Figura 6.10: Índices clase Network

6.5 Pruebas

Una de las partes fundamentales de un desarrollo es la comprobación de que el software resultante cumple con los requisitos definidos. Para este fin existen las pruebas de unidad que se caracterizan por aplicarse a una unidad. Sin embargo, mediante estas pruebas no se comprueba la calidad del software. Es por esta razón que también se somete a la librería a unas pruebas de rendimiento.

6.5.1 Entorno de pruebas

Antes de comenzar con el desarrollo de las pruebas es preciso definir dónde se van a ejecutar, pues de ello dependen las mediciones obtenidas. En este caso las pruebas se han llevado a cabo en el mismo equipo en el que se ha desarrollado el trabajo.

Modelo CPU	Intel(R) Core(TM) 7-7700HQ
Frecuencia Base/Turbo	2.80/3.80 GHz
Cache	6MB
Cores/Threads	4/8
Sistema Operativo	Debian GNU/Linux 9.11
Memoria RAM	8 GB DDR4-2400
Disco 1	SATA 1TB 7200 rpm
Disco 2	Samsung EVO 970 NVMe M2

Cuadro 6.1: Especificaciones entorno de pruebas

6.5.2 Pruebas unitarias

Comprobar que la librería cumple con los requisitos definidos es una buena práctica en el desarrollo de software pues ayuda a detectar errores cometidos durante la fase de implementación. En este caso se hace una prueba por cada módulo de interacción que se observa en la Figura 6.2

6.5.3 Pruebas de rendimiento

Bajo este tipo de pruebas se mide la capacidad de la librería para obtener el resultado de una forma eficiente. En concreto, el tipo de pruebas a la que ha sido sometida es a pruebas de estrés. En otras palabras, estas pruebas cuantifican el rendimiento bajo situaciones de alta exigencia.

Para este entorno se decide que las pruebas cubran hasta la fase de generación de configtx, crypo-config y docker-compose, previo al despliegue. El motivo de esta decisión vino

determinado porque esta última fase requiere del uso de *docker*, que no es una herramienta propia de la librería. Por esta razón, distinguiremos entre tres casuísticas:

- Variación del número de organizaciones de una red.
- Variación del número de *peers* por organización.
- Unión de ambas variaciones.

Antes de comenzar se establece un caso base sobre el que se realizaron dichas modificaciones. Se determina que la arquitectura de red común estaría compuesta por cinco organizaciones con cinco *peers* cada una. No se realizan variaciones en el número de *orderers* puesto que esta sigue una progresión lineal.

Con el fin de obtener unas mediciones fiables, la toma de tiempos se ha realizado varias veces sobre el mismo caso de prueba y se obtiene la media. De esta forma se reduce significativamente el error que se pueda cometer.

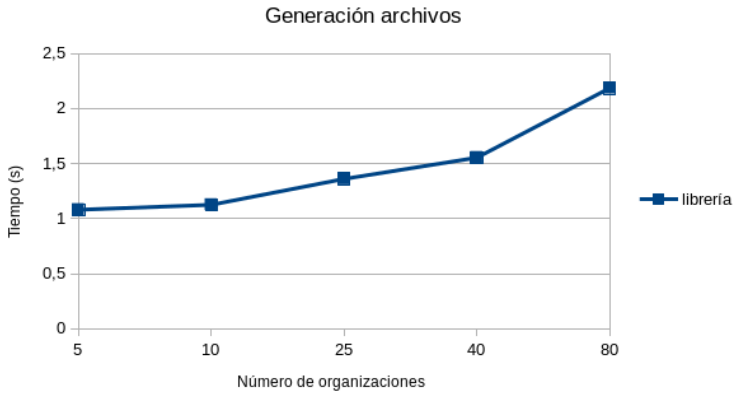

Figura 6.11: Tiempo para la generación de archivos variando las organizaciones

Como se observa en las Figuras 6.11 y 6.12 no se aprecian grandes diferencias entre ambas variaciones y los tiempos no son excesivamente grandes en los casos más exigentes lo que denota un buen rendimiento. Gracias a estas pruebas, durante la fase de desarrollo, se detectó un aumentó drástico del tiempo de ejecución en los casos más exigentes. Por medio del empleo de índices se consigue solventar esa bajada de prestaciones tal y como recoge el apartado 6.4.4.

En esta última figura se aprecia que el tiempo de ejecución aumenta drásticamente cuando se crean 80 organizaciones con 80 *peers* cada una. Esto se debe principalmente a que las pruebas sobre la librería no están paralelizadas aunque, como veremos en el capítulo 7, los tiempos disminuyen si son asíncronas.

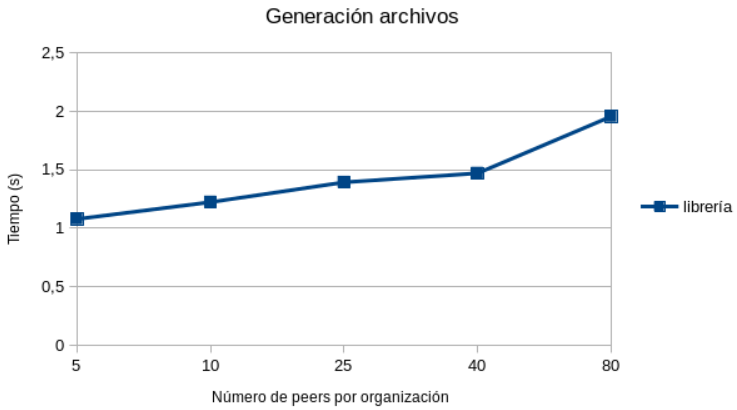

Figura 6.12: Tiempo para la generación de archivos variando los *peers*

Figura 6.13: Tiempo para la generación de archivos variando los *peers* y las organizaciones

Capítulo 7

Desarrollo de la herramienta

Una vez definido el soporte de la herramienta el siguiente paso se centra en proporcionarle una interfaz lo suficientemente clara como para que cualquier persona con conocimientos básicos pueda desarrollar una red blockchain. Este capítulo aborda todo el proceso de diseño, implementación y pruebas del software. Es aquí donde todos los conceptos vistos se unen dando como resultado la herramienta completa. Este desarrollo también utiliza Scrum como metodología con lo cual la primera tarea supone con la definición de los requisitos en el *Product Owner*. Acto seguido por medio de incrementos se fueron cumpliendo dichos requisitos. En concreto, este desarrollo cuenta con tres Sprints, uno para la implementación del *backend*, otro para el *frontend* y el tercero añade funcionalidades tanto en un lado como en el otro.

7.1 Objetivos

La finalidad de este trabajo es proporcionar una herramienta para el desarrollo de proyectos de trazabilidad por medio del uso de redes blockchain. Dicha herramienta precisa de una interfaz gráfica para agilizar el diseño de la red. Como se puede deducir, al mencionar *backend* y *frontend*, la herramienta sigue una arquitectura cliente-servidor en la que la interfaz se muestra por medio de un navegador web. Los motivos de esta elección fueron dos, por un lado el *software* debía poder ejecutarse en cualquier entorno, lo que lo haría accesible a todo el mundo y, por otro lado, las redes blockchain no suelen ejecutarse en la máquina local, sino que lo hacen en entornos en producción que generalmente no son accesibles físicamente. Un servidor remoto alberga la librería junto con todas sus dependencias y expone todas sus funcionalidades mediante una API REST. Desde la máquina cliente con un simple navegador se puede acceder a una aplicación web para el diseño y despliegue de una red blockchain.

7.2 Análisis de requisitos

Todo desarrollo software comienza por la definición de los requisitos que debe cumplir.

7.2.1 Requisitos funcionales

Los requisitos funcionales son aquellos que debe cumplir el software resultante de forma intrínseca. En este caso, los requisitos de la herramienta no difieren en gran medida de los de la librería pues es allí donde se han definido ya los casos de uso y sus requisitos. Este desarrollo expone el acceso a dicha librería por medio de una interfaz web, aún así, los requisitos se han dividido en función de la arquitectura seguida, por un lado el cliente y por otro el servidor.

- Interfaz
  - Diseñar y desplegar una red
  - Añadir, modificar y eliminar *peers*
  - Añadir, modificar y eliminar *orderes*
  - Añadir, modificar y eliminar organizaciones
  - Añadir, modificar y eliminar *channels*
  - Crear, instalar e instanciar *chaincodes*
  - Desplegar la red
- Servicio
  - Crear una red
  - Exponer las funciones relativas a los *Peers* de la librería
  - Exponer las funciones relativas a los *Orderers* de la librería
  - Exponer las funciones relativas a los *Channels* de la librería
  - Exponer las funciones relativas a los *Organizations* de la librería
  - Exponer las funciones relativas a los *Chaincodes* de la librería
  - Desplegar la red

7.2.2 Requisitos no funcionales

La completitud de los requisitos anteriormente mencionados se puede llevar a cabo de distintas maneras, todas ellas diferentes en términos de calidad del software. Es aquí donde los requisitos no funcionales, los que no están asociados con los objetivos del software final, tienen lugar. Para el desarrollo de herramienta, también se debe cumplir con los siguientes:

- Usabilidad. Referente a la facilidad con la que las personas utilizan una herramienta para alcanzar un objetivo. Este proceso debe ser intuitivo y de comprensión sencilla para que el usuario final desarrolle su propósito sin impedimentos.

- Multiplataforma. El software debe poder ejecutarse sobre cualquier infraestructura.

7.3 Diseño y planificación

Siguiendo la metodología empleada en todo el trabajo, este desarrollo también se ha dividido en iteraciones. Antes de presentar en detalle cada uno de los Sprints, se muestra una imagen global definiendo los motivos de las decisiones de diseño tanto para el *backend* como para el *frontend*.

7.3.1 Diseño general

Al igual que con la librería, antes de empezar a dividir el trabajo en incrementos es preciso tener una visión global de cómo debe ser el software. Como la aplicación final debe ser multiplataforma se optó por un modelo cliente-servidor, en el que el servidor se encargará de procesar las peticiones que le realice el cliente que estará ejecutándose sobre un navegador. Siguiendo este esquema, para el diseño es preciso distinguir entre el desarrollo de la parte servidor o *backend* y el de la parte cliente o *frontend*.

Parte Servidor o *Backend*

Para el acceso a los recursos que almacena el servidor se sigue seguir una arquitectura REST [30]. Su elección está motivada por su facilidad de uso e implantación al igual que por la experiencia adquirida durante los estudios de grado. Esta arquitectura se basa en tres puntos básicos:

- **Protocolo sin estado.** Se emplea el protocolo HTTP para la comunicación con el cliente. Una característica clave de este protocolo es que carece de estado, es decir, cada petición se procesa de la misma forma, sin tener en cuenta las anteriores. Gracias a esta función el servidor no necesita almacenar ningún histórico de las peticiones anteriores, lo que lo hace más sencillo de implementar.

- **Métodos HTTP.** Las operaciones sobre los recursos que se almacenan en el servidor siguen los métodos de dicho protocolo. GET, PUT, POST y DELETE permiten leer, editar, crear y eliminar objetos respectivamente.

- **Acceso por medio de URI's.** Una URI es una dirección que identifica de forma inequívoca cualquier recurso del servidor. Cada una de ellas ofrece el acceso, la modificación o la creación así como establece una jerarquía de los objetos

Con el objetivo de que el código resultante fuese modular y reusable se tomó la decisión de diseño de agrupar las funcionalidades de cada componente de la red en controladores, que son agrupaciones de funciones con alta cohesión encargadas de procesar las peticiones al servidor. De esta forma todas las operaciones sobre los *peers*, las procesará un único manejador.

Parte Cliente o *Frontend*

En el caso de la interfaz, dado que la decisión de emplear Angular 7 estaba tomada, se divide la interfaz siguiendo su arquitectura, como se muestra en la Figura 7.1. Se desarrolla un módulo para cada elemento de la red (*peer*, *orderer*, *organization*, *chaincode* y *network*), y a su vez cada uno de ellos incorpora un servicio que es el encargado de realizar las peticiones al servidor y uno o más componentes, que hacen de intermediarios entre el servicio y la plantilla HTML.

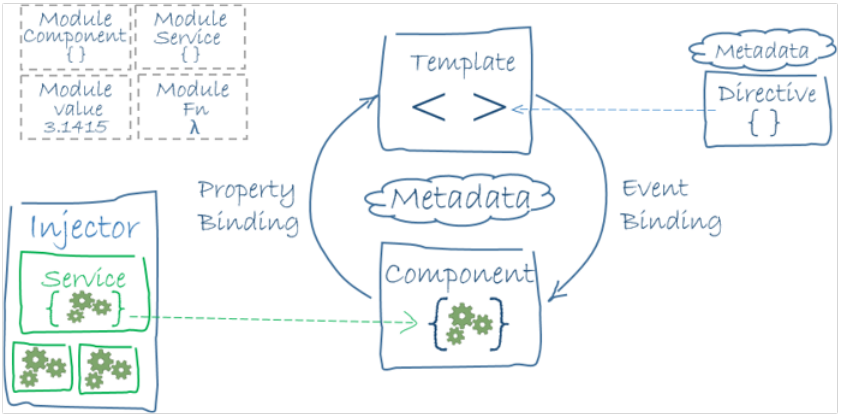

Figura 7.1: Arquitectura Angular

Dentro de la interfaz distinguiremos dos entornos. Tras iniciar la herramienta entraremos en un modo diseño de la red, donde se definen los *peers*, los *orderers* y demás componentes de la red. Una vez finalizado el diseño y desplegada la red, el entorno cambia y permite el desarrollo de *chaincodes*, su instalación en canales y la generación de plantillas.

7.3.2 Sprint 5

El diseño de la herramienta comienza con la definición de los controladores que atenderán las peticiones y sus *endpoints* asociados. Cada manejador interactúa únicamente con un mó-

dulo de interacción que expone la librería, esto proporciona un bajo acoplamiento al software resultante, lo que facilita su reutilización y mantenimiento.

Una vez diseñados los controladores, en el siguiente paso se asocia cada función con su *endpoint* y su método HTTP siguiendo la arquitectura RESTful.

- /network
  - GET. Obtiene el dominio de la red
  - POST. Crea una nueva red
  - DELETE. Elimina la red
- /channels
  - /
    - GET. Obtiene todos los canales
    - POST. Crea un nuevo canal
  - /:channelid
    - GET. Obtiene el canal con un determinado identificador
    - PUT. Edita el canal con un determinado identificador
    - DELETE. Elimina el canal con un determinado identificador
- /orgs
  - /
    - GET. Obtiene todas las organizaciones
    - POST. Crea una nueva organización
  - /:orgid
    - GET. Obtiene la organización con un determinado identificador
    - PUT. Edita la organización con un determinado identificador
    - DELETE. Elimina la organización con un determinado identificador
  - /:orgid/peers
    - GET. Obtiene todos los *peers* de una organización
    - POST. Añade un nuevo *peer* a la organización
    - DELETE. Elimina la organización con un determinado identificador
  - /:orgid/peers/:peerid
    - GET. Obtiene la el pee con un determinado identificador en su organización
    - PUT. Edita el *peer* con el identificador en su organización

        * DELETE. Elimina un determinado *peer* dentro de una organización

- /orderers
  - /
    * GET. Obtiene todos los *orderers*
    * POST. Crea un nuevo *orderer*
  - /:ordererid
    * GET. Obtiene el *orderer* con un determinado identificador
    * PUT. Edita el *orderer* con un determinado identificador
    * DELETE. Elimina el *orderer* con un determinado identificador

7.3.3 Sprint 6

Con este incremento se inicia el desarrollo de la interfaz web. Angular 7 fue el framework escogido para su desarrollo, debido a esto el diseño se ajustó a la arquitectura que ofrece dicho framework es su documentación. En un primer momento se crean los *mockups* o maquetas de las distintas vistas de cada entorno que incorporar la interfaz tal y como se puede observar en las Figuras 7.2. El resto se pueden encontrarse en el Apéndice A

Una vez terminados, el siguiente paso fue dividir en módulos y componentes Angular los distintos elementos de los *mockups*, distinguiendo entre los comunes y los específicos de las vistas. Como resultado surgieron seis módulos, cinco de ellos para las vistas específicas compuestos por un componente angular y un servicio para interactuar con el servidor, el módulo restante es el que está formado únicamente por un componente y emplea los servicios de los otros módulos.

Por otro lado, se diseñaron unas clases modelo que definen los componentes de la red blockchain y que son usadas para el intercambio de información entre los componentes Angular y sus servicios.

Objetivos

Los objetivos de ente Sprint fueron:

- Diseño e implantación de las vistas de los *peers*.
- Diseño e implantación de las vistas de los *orderers*.
- Diseño e implantación de las vistas de las organizaciones.
- Diseño e implantación de las vistas de los canales.
- Diseño e implantación de las vistas de los *chaincode*.

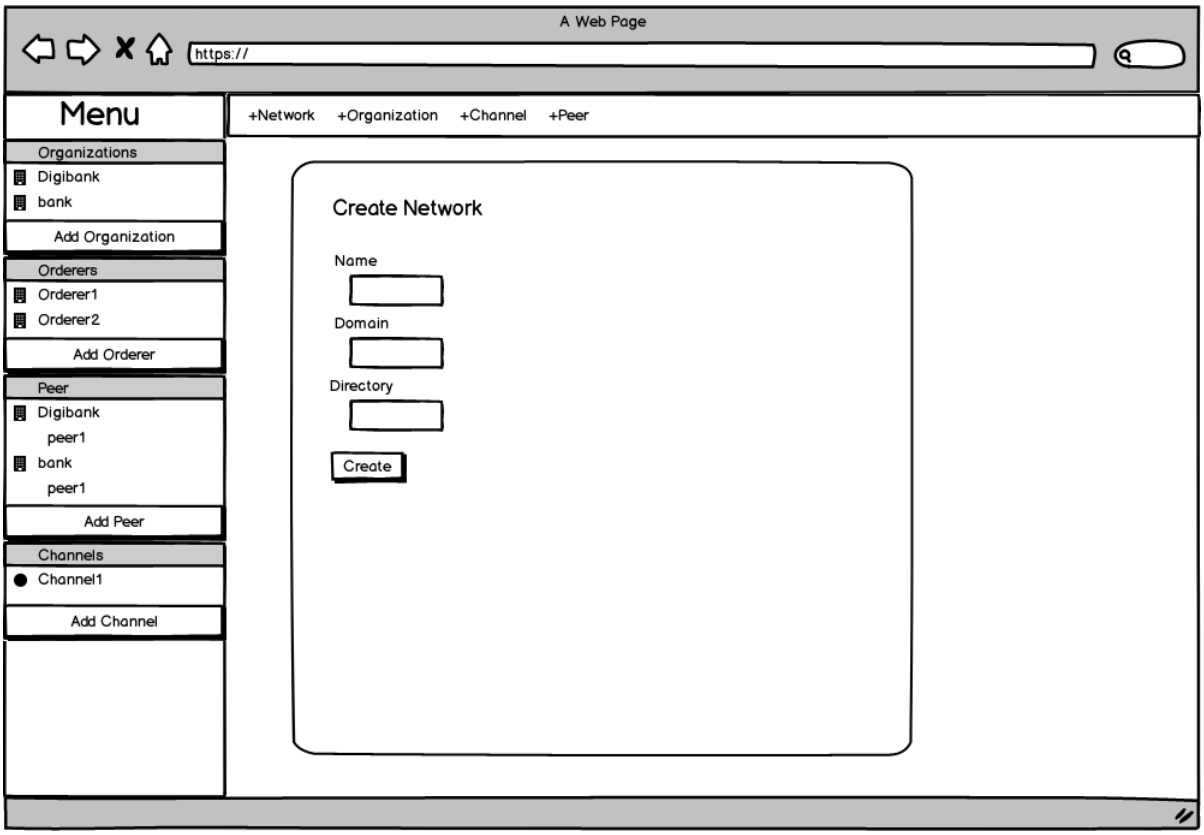

Figura 7.2: *Mockup* de la vista crear red

7.3.4 Sprint 8

La última iteración de todo el trabajo consistió en incorporar a la herramienta el empleo de *chaincodes* y la generación de plantillas de conexión con la red blockchain para dispositivos IoT. Para ello se diseña un ChaincodeController en el backend que sería el encargado de enviar las peticiones a la librería. Después vendría el diseño de los mockups de las vistas que mostrarían las funciones del controlador, como se puede observar en la Figura 7.4 y en el Apéndice A. Por último, se implementaron dichas vistas dividiéndolas en componentes Angular tal y como se hizo en el Sprint 6.

Objetivos

Los objetivos de ente Sprint fueron:

- Diseño e implantación de las vistas de los *chaincodes*.
- Diseño e implementación del controlador de los *chaincodes*.

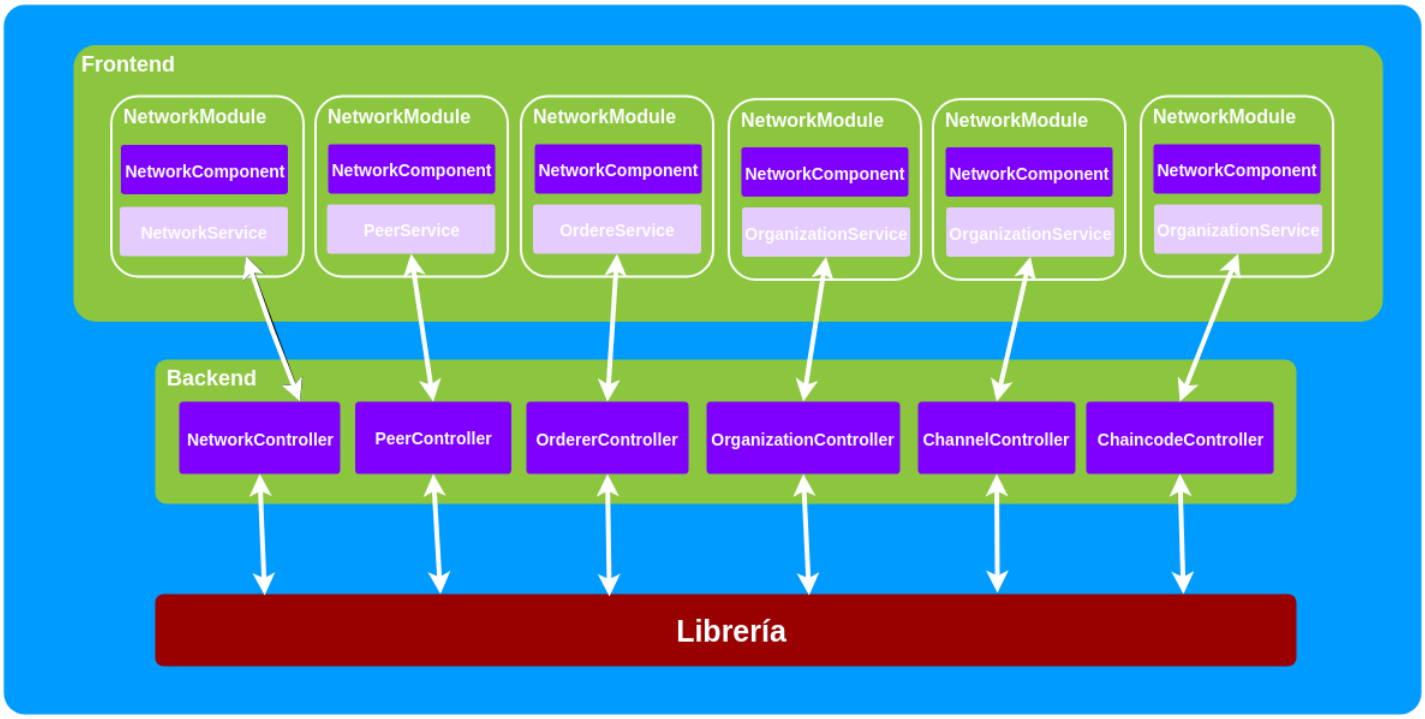

Figura 7.3: Diagrama de la librería

7.4 Implementación

Este apartado engloba todo el proceso de codificación destacando lo más relevante. Con la finalidad de una mayor comprensión esta parte se ha dividido en la parte servidor en dos grandes apartados, parte servidor o *backend* por un lado y, por otro, parte cliente o *frontend*. Al igual que con la librería

7.4.1 *Backend*

Para agilizar el desarrollo de la parte servidor se utiliza el framework Espress, que como ellos afirman «es una infraestructura de aplicaciones web Node.js mínima y flexible que proporciona un conjunto sólido de características para las aplicaciones web» [19]. Entre todos los beneficios que aporta destacan dos:

- Creaión rápida de *endpoints*.
- Uso de *middlewares* o funciones que se encargan de manejar las peticiones al servidor. Gracias a ellos podemos redirigir las solicitudes en función de su método HTML a su manejador correspondiente.

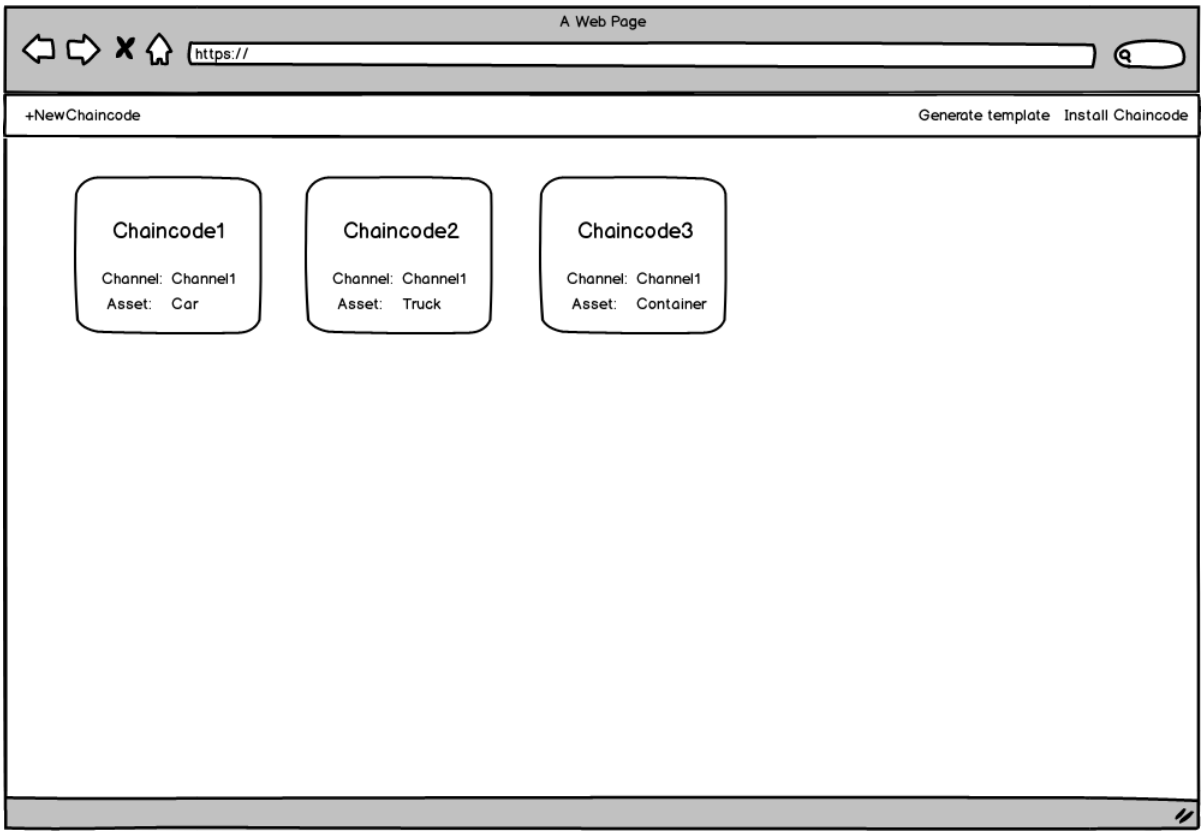

Figura 7.4: *Mockup* del segundo entorno de la interfaz

Controladores

Como se comentó en el apartado 7.3.2 de diseño, el código se ha dividido en controladores, de esta manera se reduce el acoplamiento y aumente la cohesión. Cada manejador funciona de intermediario entre el servidor y la librería y tienen una estructura similar a la de la Figura 7.6. La herramienta cuenta con cinco manejadores:

- *NetworkController*, es el encargado de procesar las peticiones que están asociadas al control de la red. Permite crear, eliminar construir y desplegar la red.
- *PeerController*, maneja las funciones relativas a los *peers*. Implementa las funciones básicas de crear, modificar y eliminar *peers*. Pero también permite obtener peer por identificador y obtener los *peers* de una determinada organización.
- *OrdererController*, muy similar al *PeerController* pero sobre *orderers*. Se encarga de la creación, actualización y eliminación además de la consulta de uno o todos los *orderers*.
- *OrganizationController*, expone las funciones de la librería referentes a las organizaciones. Permite realizar las operaciones básicas de creación, modificación y eliminación,

```
orgs
  .route("/:orgId")
  .put(OrgCtrl.updateOrg)
  .get(OrgCtrl.findOrg)
  .delete(OrgCtrl.deleteOrg);

orgs
  .route("/:orgId/peers/:peerId")
  .get(PeerCtrl.findPeer)
  .put(PeerCtrl.updatePeer)
  .delete(PeerCtrl.deletePeer);

orgs
  .route("/:orgId/peers")
  .get(PeerCtrl.findAllPeersForOrg)
  .post(PeerCtrl.createPeer);

channels
  .route("/:channelId")
  .get(ChannelCtrl.findChannel)
  .put(ChannelCtrl.updateChannel)
  .delete(ChannelCtrl.deleteChannel);

channels
  .route("/")
  .get(ChannelCtrl.findAllChannels)
  .post(ChannelCtrl.createChannel);

app.use("/orgs", orgs);
app.use("/channels", channels);
app.use("/orderers", orderers);
app.post("/directory", NetworkCtrl.setDestDirectory);
app.post("/build", NetworkCtrl.build);
app.post("/launch", NetworkCtrl.launch);
```

Figura 7.5: ExpressJS

aunque también es posible consultar organizaciones por su identificador u obtener todos los *peers* que la componen.

- *ChaincodeController*, aglutina todas las funciones referentes a los *chaincodes*. Permite instalar, instanciar, consultar e invocar. También ofrece la funcionalidad de generar los archivos para la interacción de un dispositivo IoT con la red.

Error Manager

Los controladores trabajan directamente con la librería, pudiéndose producir errores, tal y como vimos en el capítulo 6. Estos se propagan al servidor pudiendo llegar a ocasionar una caída del servicio. *Error Manager* es el módulo encargado de capturar los distintos fallos que se puedan producir y responder a la petición con un mensaje de error. Cuando se produce un fallo, los controladores delegan en este módulo su gestión. Como todos los errores de la librería pertenecen a la clase *ErrorWithCode*, el *ErrorManager* simplemente tiene que comprobar si el fallo pertenece a esa instancia. En caso afirmativo devuelve una respuesta al cliente con el

```
const ModelOrderer = require('../../src/models/modelOrderer.js')
const ErrorWithCode = require('../../lib/error/error')
const Errors = require('../utils/errorManager');
//GET - Return all Orderers in the network
exports.findAllOrderers = function(req, res) {
    try {
        res.status(200).send(ModelOrderer.getAllOrderers())
    }catch(err){
        Errors.errorManager(res,err);
    }
};

//GET - Return a Orderer with specified ID
exports.findOrderer = function(req, res) {
    try {
        res.status(200).send(ModelOrderer.getOrderer(req.params.ordererId))
    }catch(err){
        Errors.errorManager(res,err);
    }
};

//POST - Insert a new Orderer in the Network
exports.createOrderer = function(req, res) {
    try{
        //|| typeof req.body.extPort !== 'number'
        if (req.body.name ==null ||req.body.name=='' || req.body.id== null ||req.body.id== ''|| req.body.extPort==null ){
            console.log(req.body.name +" "+  req.body.id +" "+ req.body.extPort)
            throw new ErrorWithCode(400,"Name, Identification and ExtPort required")      You, 6 months ago • Test backend Orderers Fini
        }else{
            res.status(200).send(ModelOrderer.createOrderer(req.body.name,req.body.id,req.body.extPort,req.body.intPort,req.body.extra))
        }
    }catch(err){
        Errors.errorManager(res,err);
    }

};
```

Figura 7.6: Implementación *OrdererController*

código de estado HTML del error y su descripción, en caso contrario, devuelve una petición con el código de estado 500.

7.4.2 *Frontend*

Como recoge el apartado 7.3.3, la implementación de la interfaz se realizo con el framework Angular 7. Por este motivo, el diseño estuvo altamente influenciado por su arquitectura basada en componentes, servicios y plantillas. Los componentes son las piezas fundamentales dentro del framework pues implementan toda la lógica de la vista. Actúan como intermediarios entre las plantillas HTML y las clases que acceden al servicio [23].

ToastrService

Muchas de las operaciones que realiza la librería no tienen una respuesta inmediata. Por esta razón la herramienta incorpora un servicio de notificación para comunicar al usuario que una operación ha finalizado. Esta funcionalidad se incorpora por medio la de la instalación de

un módulo con un servicio que recibe peticiones y muestra por pantalla un *toast* o notificación. Dependiendo de si la operación ha terminado correctamente o no el mensaje mostrado varía al igual que el color.

Haciendo uso de este servicio se implementan dos clases que facilitan el uso de la librería. Los métodos *successManager* y *errorManager* son accesibles a todos los componentes Angular de la aplicación web, permitiendo mostrar notificaciones de éxito o error respectivamente como se observa en la Figura 7.7

```
import { ToastrService } from "ngx-toastr";

export function errorManager(toaster: ToastrService, err:any){
    if (err.error!=null&&err.error.error_message!=null)
        toaster.error("Response code: "+err.status,err.error.error_message,
        {timeOut: 3000})
    else{
        if (err.status==500){
            toaster.error("Response code: "+err.status,"Fatal Error",
        {timeOut: 3000})
        }else{
            toaster.error("Response code: "+err.status,"Error with server",
        {timeOut: 3000})
        }
    }

}

export function successManager(toaster: ToastrService, title:string, subtitle:string){
    toaster.success(subtitle,title,
        {timeOut: 2000})
}
```

Figura 7.7: Implementación del servicio de notificaciones

Componentes

En base a los *mockups*, en la interfaz de la herramienta se pueden distinguir dos elementos comunes, por un lado el menú lateral y la barra de navegación y, por otro, un sección que va cambiando en función de los datos seleccionados. Por esta razón se implementa un componente para el menú lateral y otro componente por cada elemento de la red que agrupe el formulario de creación y el de actualización. La función de estos últimos es comprobar que los datos introducidos cumplen con los requisitos que se necesitan, es decir, que no falte el campo identificador si es necesario, por ejemplo.

Templates

Como se puede observar en el Apéndice A, las vistas para añadir un nuevo componente a la red son muy similares a las que muestran la información de dicho componente. Implementar dos plantillas prácticamente iguales con campos deshabilitados sería una mala práctica, pues el código sería redundante. Una mejor opción, que fue la implantado, consiste en crear una única plantilla con atributos que dependen de una variable que se controla desde el componente Angular asociado. De esta manera el código resultante resulta más legible y menos redundante.

```
onSubmit() {
  this.submitted = true;
  if (this.registerForm.valid && !this.registerForm.pending) {
    console.log(this.registerForm.value);
    this.oredererService.addOrderer(this.registerForm.value).then(
      data => {
        successManager(
          this.toastr,
          "Orderer " + this.registerForm.value.name + " created",
          null
        );
      },
      err => {
        errorManager(this.toastr, err);
      }
    );
  } else {
    console.log("Form is invalid");
  }
}

onUpdate() {

  console.log(this.registerForm.value)
  this.oredererService.updateOrderer(this.ordererId,this.registerForm.value)
  console.log("Updating");
}
onDelete() {
  console.log("Deleting "+this.ordererId)
  this.oredererService.deleteOrderer(this.ordererId)

}
```

Figura 7.8: Implementación *OrdererComponent*

7.5 Pruebas

Siguiendo con la metodología ágil, para cada uno de los incrementos se realizaron sus correspondientes pruebas. Al igual que en la librería estas se han dividido en dos grupos: unitarias y de rendimiento.

Para las mediciones del rendimiento es preciso conocer las características del equipo en el que se ejecutan. En este caso es el mismo que en de la librería, por lo que se pueden encontrar en su capítulo en el apartado 6.5.1.

7.5.1 Pruebas unitarias

Estas pruebas pretenden encontrar errores que se hayan podido cometer durante el desarrollo de alguna de la unidades del software, sometiendo al código a situaciones no previstas en un primer momento. En este trabajo, cada controlador se ha sometido a estas pruebas.

Las pruebas en este caso se realizaron con la librería Chai, que permite la ejecución asíncrona y está basada en aserciones, y el framework Moka, que facilitan la comprensión de los casos de prueba por su similitud con el lenguaje natural. Las pruebas se han dividido en seis módulos, uno por cada controlador, llegando a comprobar más de 140 casos de prueba. En la Figura 7.9 se observa un ejemplo.

```
it('should create a peer2 in org1', (done) => {

    chai.request(url)
        .post('/orgs/'+org.orgId+'/peers')
        .send(newPeer('peer2',org.orgId,config1))
        .end( function(err,res){
            expect(res).to.have.status(200)
            expect(res.body).to.have.property('PeerId').equal('peer2')
            expect(res.body).to.have.property('Domain').equal('org1.miredseg.com')
            expect(res.body).to.have.property('IntPort').equal(7050)
            expect(res.body).to.have.property('IntGossipPort').equal(7063)
            expect(res.body).to.have.property('ExtGossipPort').equal(7063)
            expect(res.body).to.have.property('isAnchor').equal(true)
            done()
        })
})
```

Figura 7.9: Test de unidad sobre la clase *Peer*

7.5.2 Pruebas de estrés

Las pruebas de estrés tienen como objetivo medir el rendimiento de un desarrollo bajo situaciones de alta exigencia. En nuestro caso, buscamos medir la calidad del *backend* sometiéndolo a pruebas con una carga de trabajo incremental. Se decide emplear los mismos casos de prueba que en la librería para poder comparar los resultados y determinar cuanto tiempo supone el uso del *backend*.

Al igual que en la librería, con el fin de obtener unas mediciones fiables, estas pruebas se han realizado bajo como una donde la ejecución exclusiva de la herramienta así como también la toma de tiempos en distintos momentos.

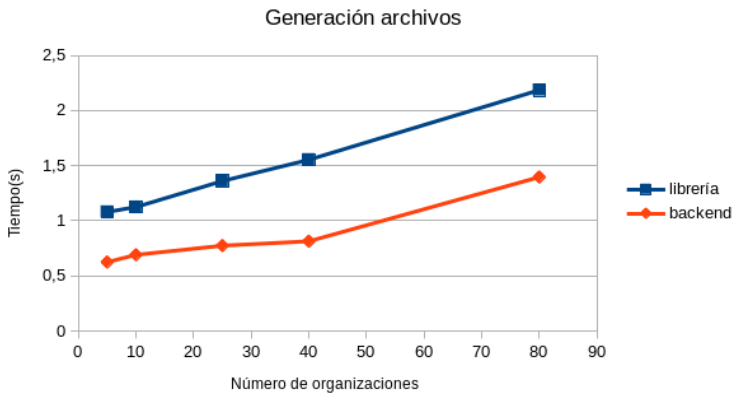

Figura 7.10: Tiempo para la generación de archivos variando las organizaciones

Como se observa en las Figuras 7.10 y 7.11, la diferencia entre los resultados obtenidos por la librería se debe principalmente a dos factores. Por un lado, al ejecutar peticiones sobre el *backend* la librería ya está cargada en el servidor, mientras que al realizar los casos de prueba

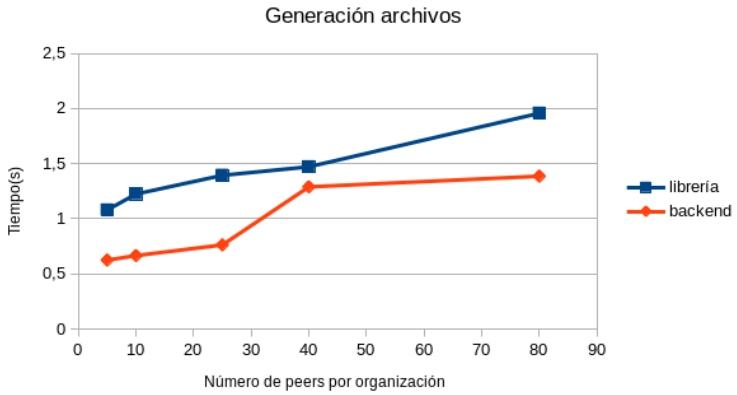

Figura 7.11: Tiempo para la generación de archivos variando los *peers*

sobre la librería primero se cargan las clases y luego se ejecutan las pruebas. Por otro lado, para los tests realizados sobre el *backend* se ha empleado la librería Chai, que permite ejecutar consultas de forma asíncrona, lo que reduce notablemente los tiempos cuanto mayor sea la carga de trabajo.

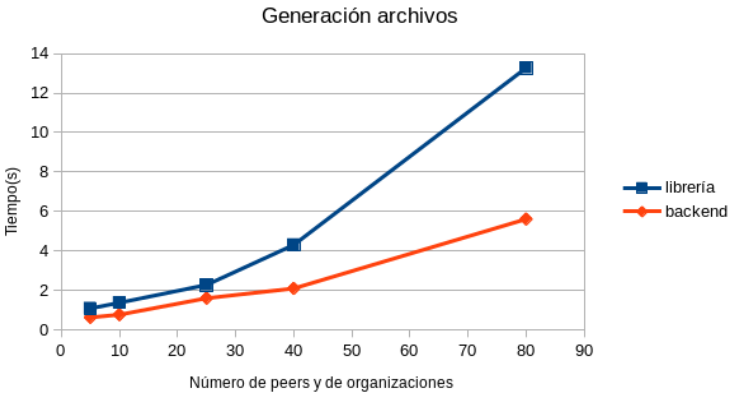

Figura 7.12: Tiempo para la generación de archivos variando los *peers* y las organizaciones

Capítulo 8

Conclusiones y trabajo futuro

El objetivo de este trabajo ha sido el desarrollo de una herramienta que permita el diseño y despliegue de redes blockchain con la finalidad de realizar la traza sobre cualquier objeto que se especifique. Este capítulo recoge las conclusiones a las que se ha llegado tras realizar dicho trabajo y las funciones que se pueden añadir a la herramienta.

8.1 Conclusiones

La principal motivación de este trabajo es la de reducir la complejidad de la tecnología Blockchain y acercarla a los desarrolladores a través de un ámbito para el que dicha tecnología sí aporta un valor añadido, la trazabilidad. En ese sentido, la herramienta reduce significativamente la dificultad gracias a la automatización tanto del despliegue como de la generación del contenido necesario. Además, por las pruebas realizas (capítulos 6 y 7) se puede determinar que el software resultante tiene calidad, pues cumple los requisitos definidos y lo hace de forma eficiente.

Con este trabajo he podido profundizar en la tecnología Blockchain por medio de la comunidad y las distintas herramientas que existen. También han surgido nuevas ideas y proyectos para desarrollar en un futuro. Respecto a los conocimientos adquiridos durante el grado, este proyecto me ha ayudado a afianzar conceptos como el desarrollo siguiendo una metodología ágil, patrones de diseño software, o seguridad a nivel de red.

Por otro lado, también cabe destacar las prestaciones del software creado. (1) Compatibilidad con múltiples entornos de producción pues implementa la arquitectura cliente-servidor. (2) Coste de implantación prácticamente nulo (una vez instalado Hyperledger Fabric), ya que el gestor de dependencias NPM se encarga de instalar todos los paquetes necesarios. (3) Simplificación del proceso de diseño y despliegue de una red blockchain, lo que supone un ahorro en tiempo y recursos. (4) Específicamente diseñada para aplicar la trazabilidad a múltiples entornos.

8.2 Trabajo futuro

El diseño de esta herramienta está pensado para poder añadir nuevas funcionalidades. Por esta razón sigue una arquitectura modular, como se aprecia en las Figuras 6.2 y 7.3, lo que permite agregar nuevos componentes de forma sencilla. Entre las funciones que se pueden añadir, destacaría tres: (1) definición de número de usuarios para cada organización. Actualmente la herramienta no permite establecer el número de usuarios que formarán parte de la red. La dificultad de incorporar dicha funcionalidad no es muy alta, pero se sale del alcance del trabajo. (2) Especificar la *endorsement policy*. La herramienta establece la misma política para todos sus *chaincodes* pues implementar dicha funcionalidad implicaría demasiado esfuerzo. (3) Mejora de la interfaz. Aunque la actual sea sencilla puede no resultar intuitiva para todo el mundo, es por esto que podría realizarse un estudio con usuario finales para conocer cual sería el flujo de trabajo que mejor se adapta a más usuarios.

En la actualidad el sector empresarial de las tecnologías de la información y las comunicaciones está inmersa en el desarrollo de diversos proyectos de trazabilidad en multitud de dominios de aplicación. Las posibilidades de la herramienta desarrollada permite un fácil y rápido despliegue de la tecnología en cualquier campo, y con unos costes reducidos.se está analizando la viabilidad de la explotación de la misma en el ámbito empresarial.

Apéndices

Apéndice A

Mockups de la interfaz

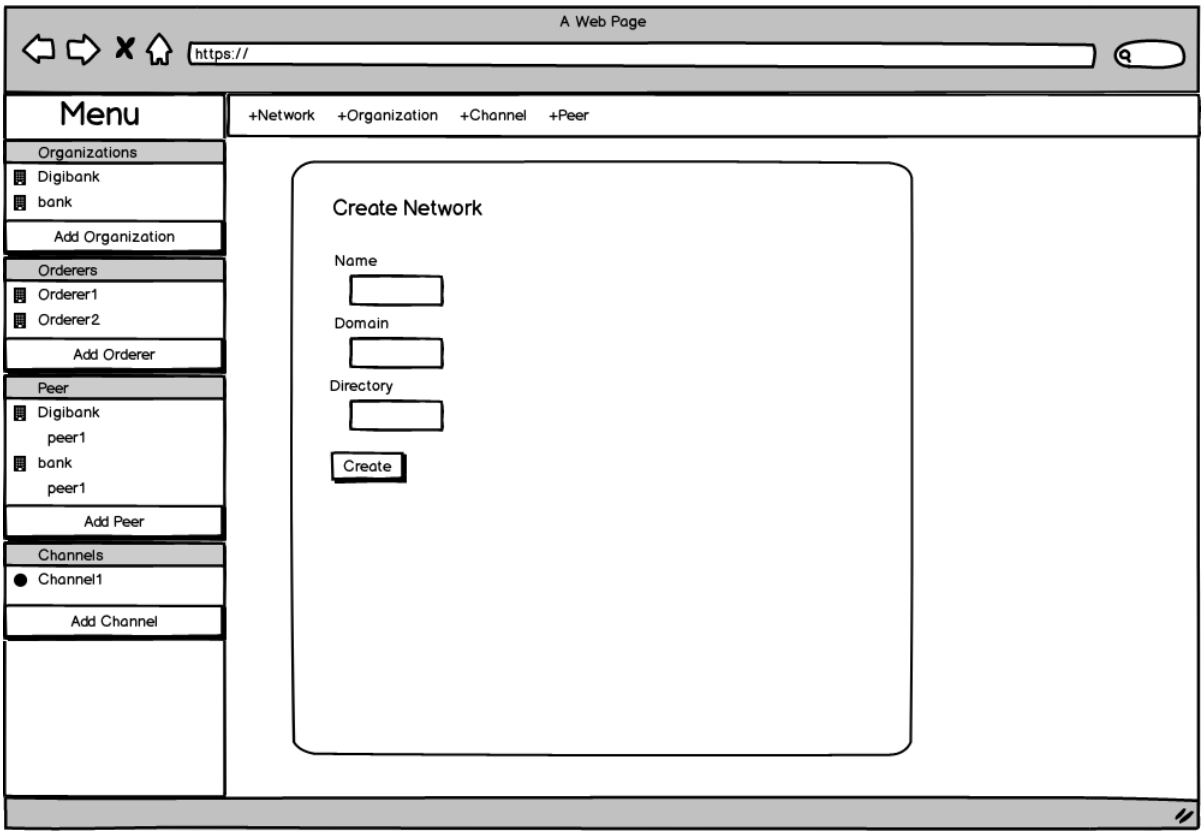

Figura A.1: *Mockup* de la vista crear red

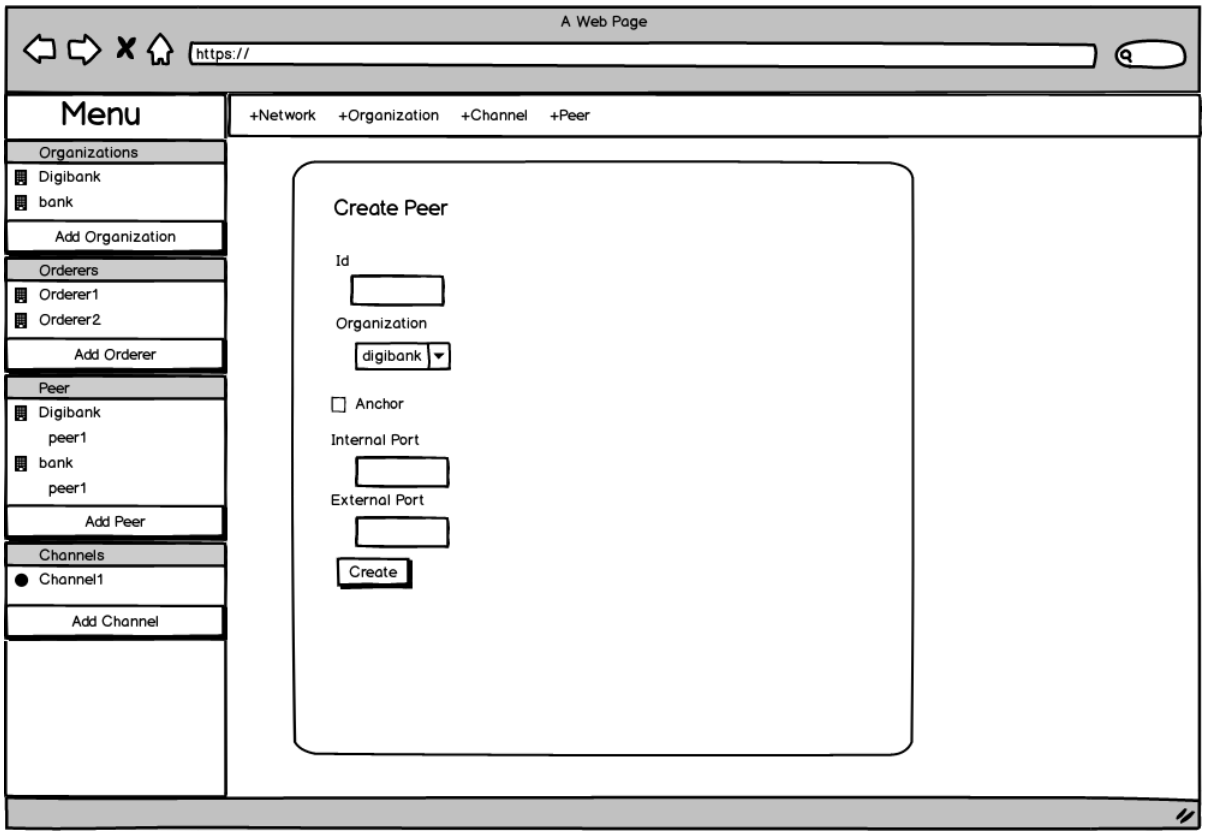

Figura A.2: *Mockup* de la vista crear un *peer*

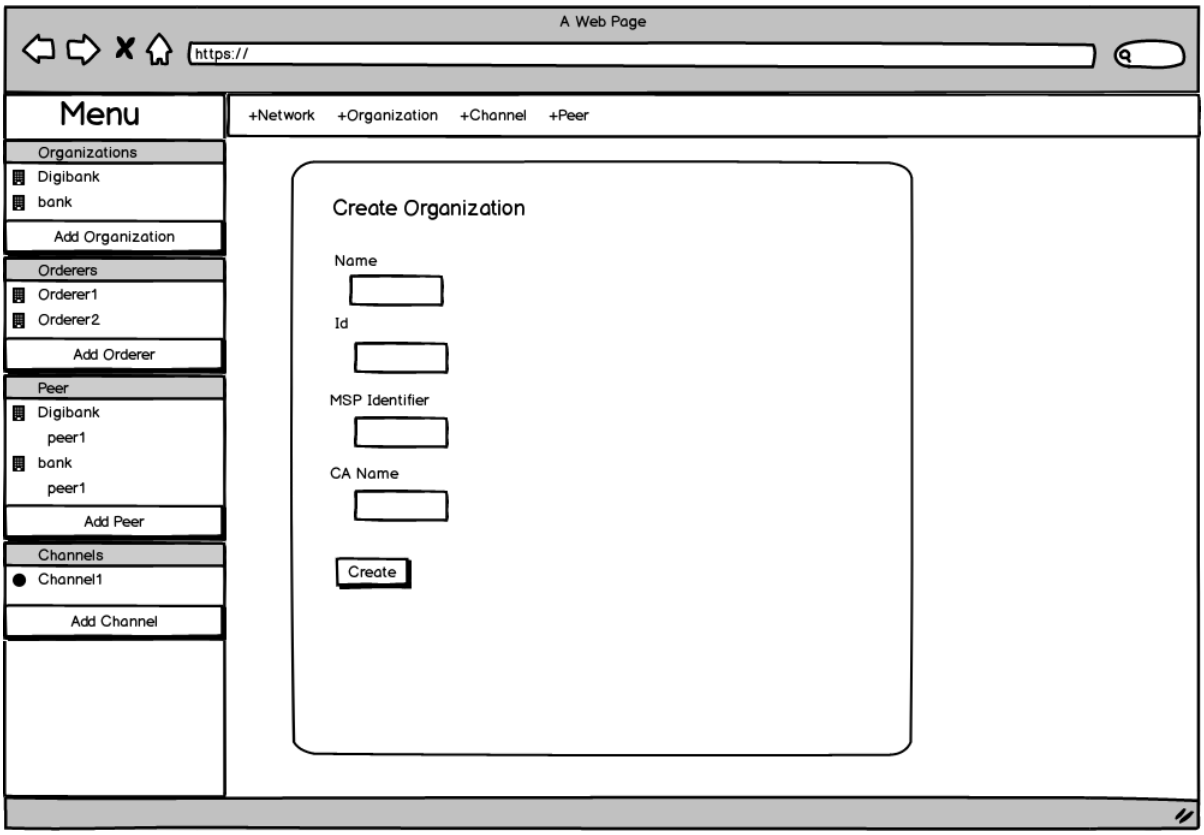

Figura A.3: *Mockup* de la vista crear una organización

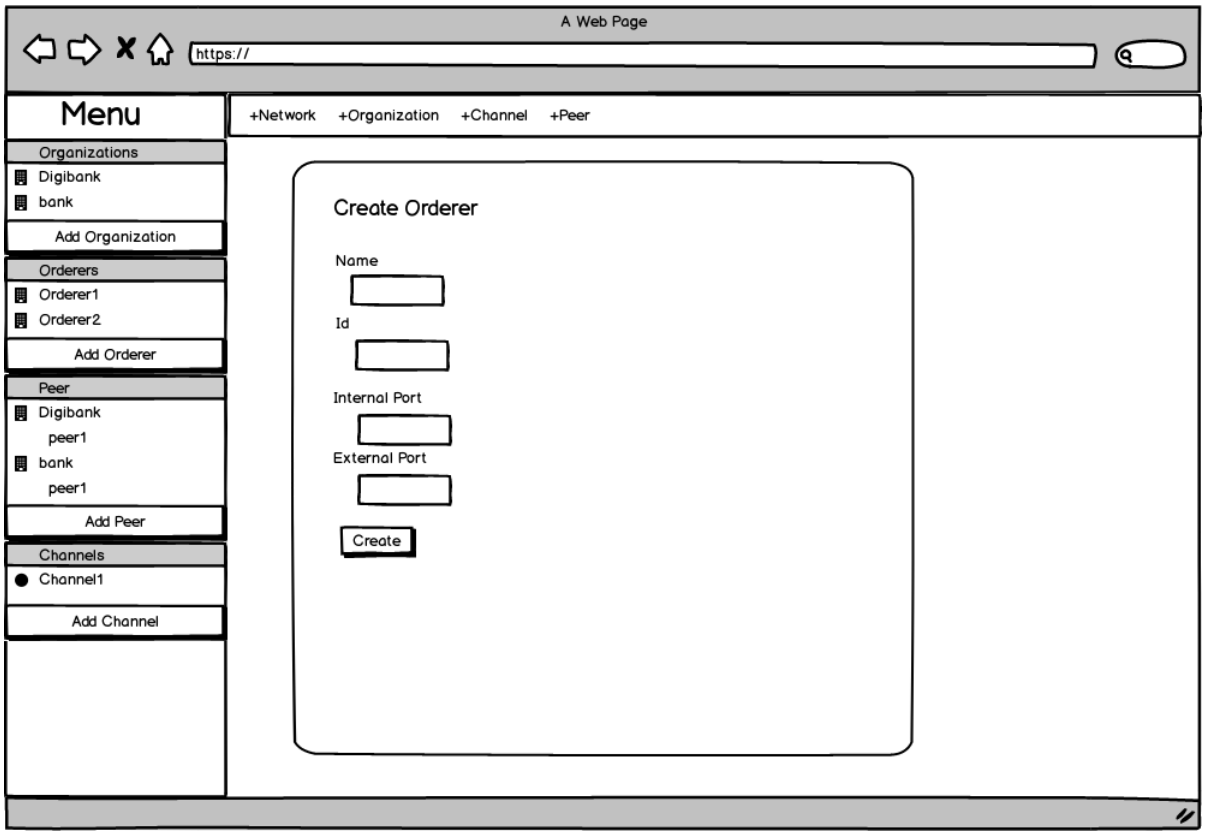

Figura A.4: *Mockup* de la vista crear un *orderer*

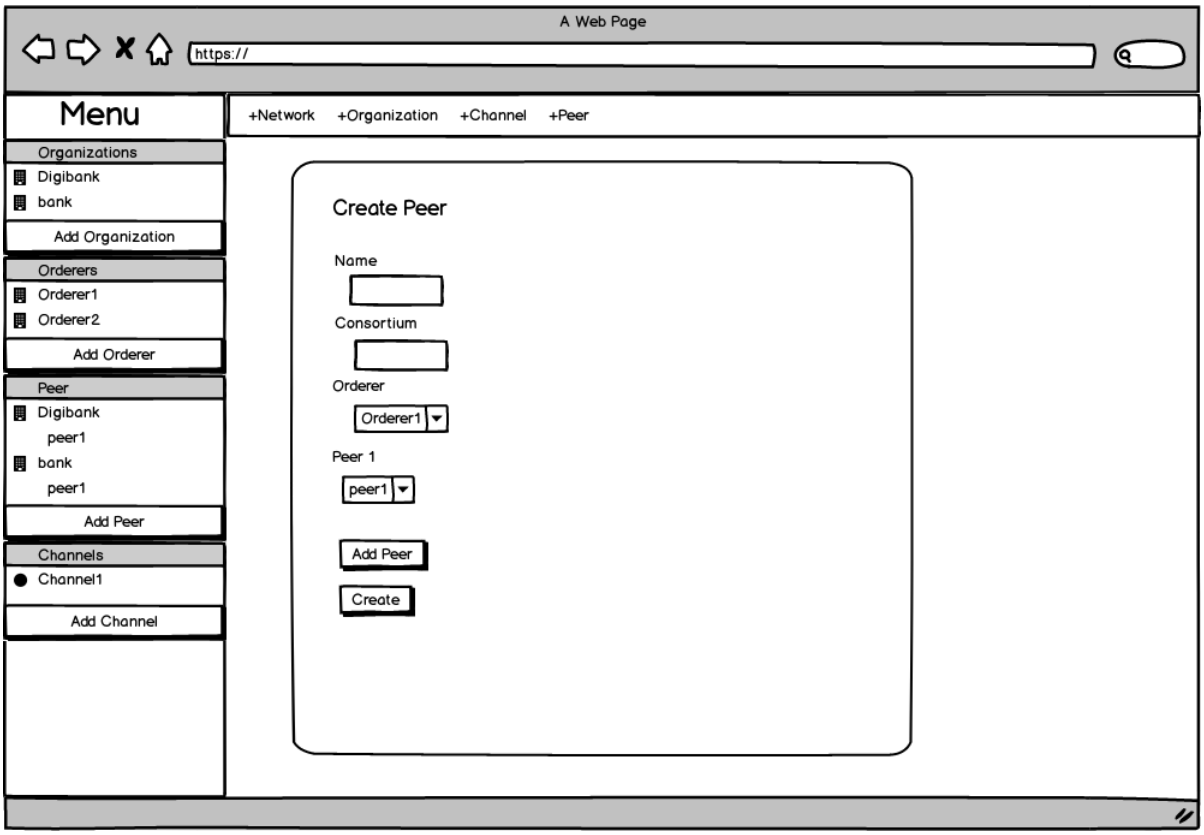

Figura A.5: *Mockup* de la vista crear un canal

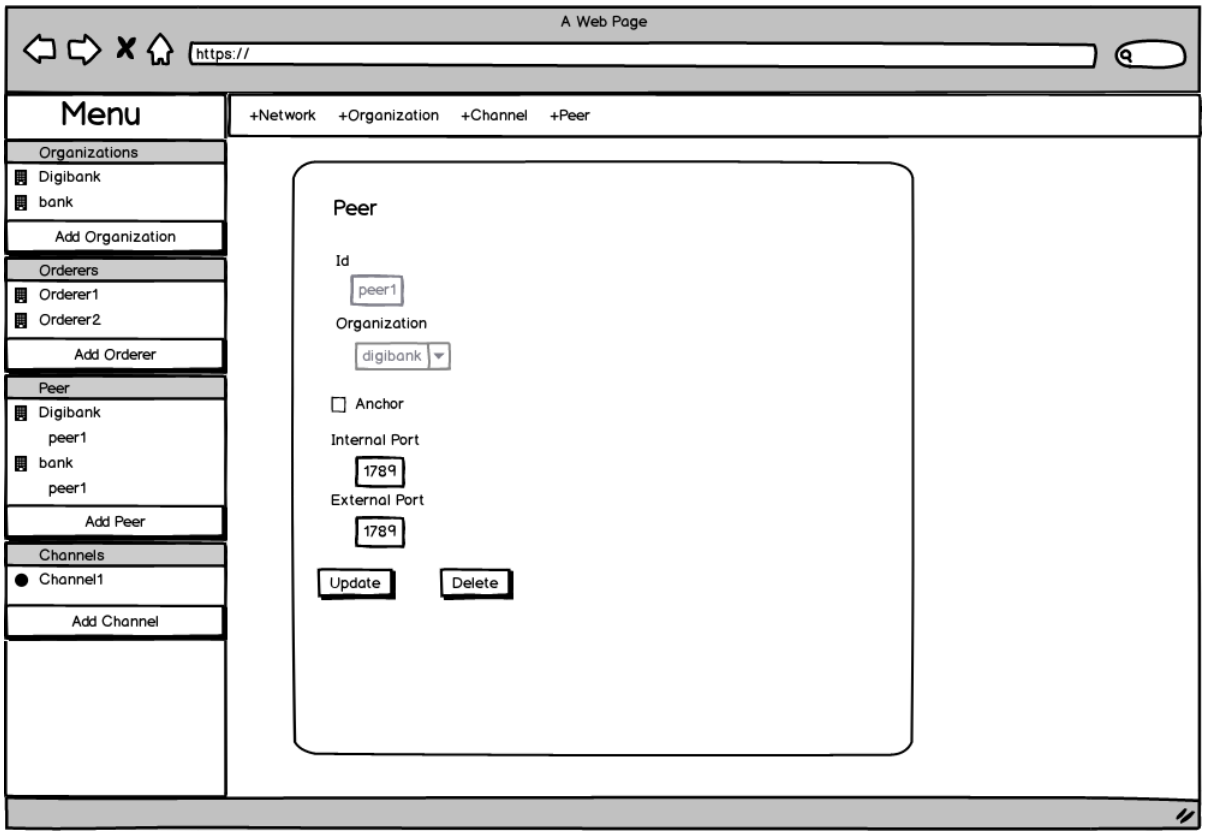

Figura A.6: *Mockup* de la vista editar un *peer*

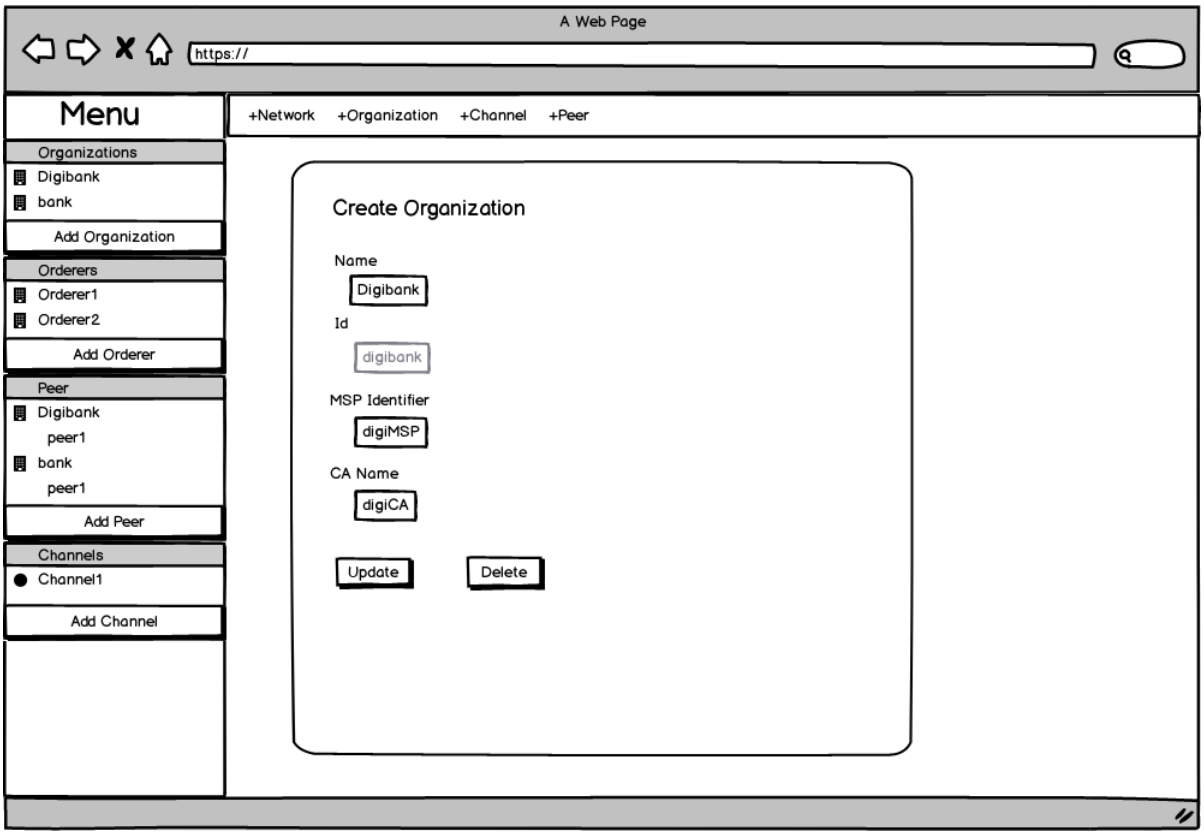

Figura A.7: *Mockup* de la vista editar una organización

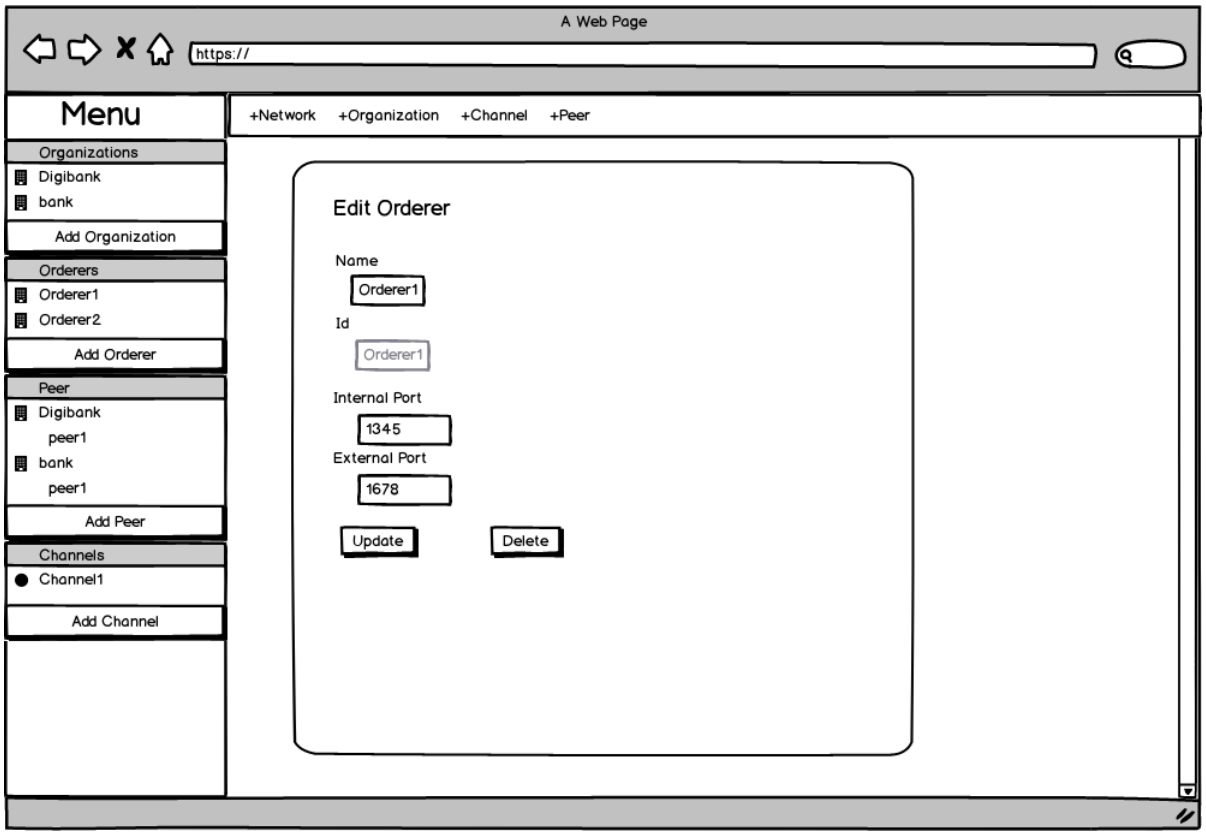

Figura A.8: *Mockup* de la vista editar un *orderer*

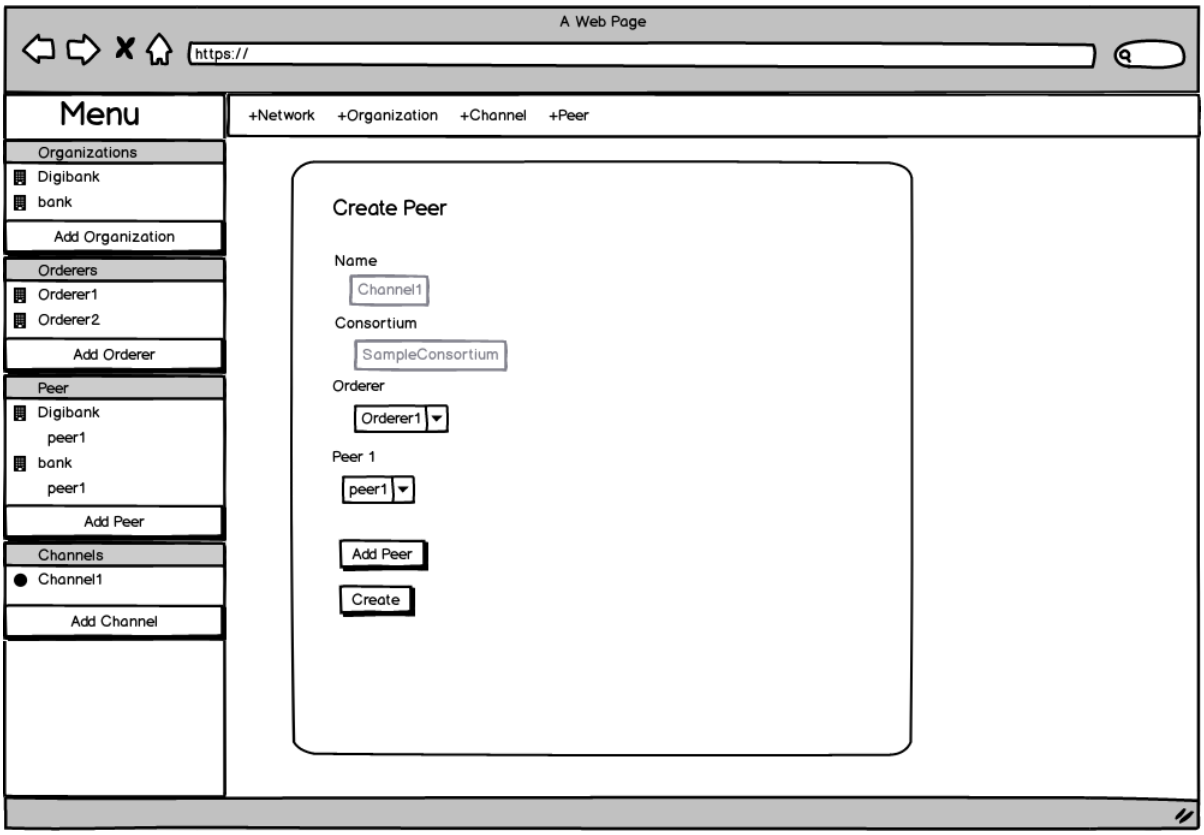

Figura A.9: *Mockup* de la vista editar un canal

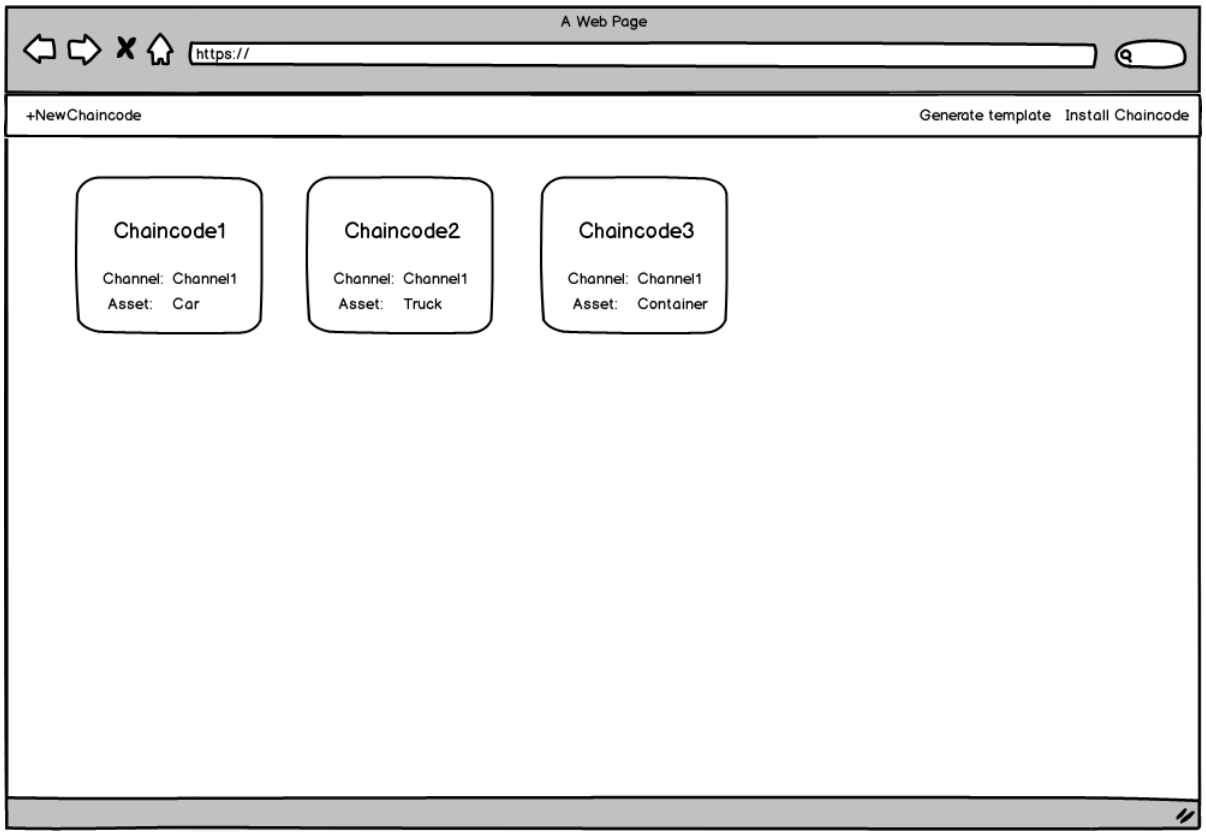

Figura A.10: *Mockup* del segundo entorno de la interfaz

Apéndice B

Glosario de acrónimos

IoT *Internet of Things.*

SDK *Software Development Kit.*

MSP *Membership Service Provider.*

AC *Autoridad Certificadora.*

Apéndice C

Glosario de termos

Smart-contract. Es un programa informático que facilita, asegura, hace cumplir y ejecuta acuerdos registrados entre dos o más partes (por ejemplo personas u organizaciones).

Red blockchai. Es una estructura de datos en la que la información se agrupa en conjuntos (bloques) a los que se les añade metainformaciones relativas a otro bloque de la cadena anterior en una línea temporal, la información contenida en un bloque solo puede ser repudiada o editada modificando todos los bloques posteriores.

Framework. O marco de trabajo, es un conjunto estandarizado de conceptos, prácticas y criterios para enfocar un tipo de problemática particular que sirve como referencia, para enfrentar y resolver nuevos problemas de índole similar.

Bibliografía

[1] ISO 9000:2015(en), quality management systems — fundamentals and vocabulary. [Online]. Available: https://www.iso.org/obp/ui/#iso:std:iso:9000:ed-4:v1:en:term:3.6.13

[2] "Commission implementing regulation (EU) no 931/2011 of 19 september 2011 on the traceability requirements set by regulation (EC) no 178/2002 of the european parliament and of the council for food of animal origin text with EEA relevance." [Online]. Available: http://data.europa.eu/eli/reg_impl/2011/931/oj/eng

[3] Food safety traceability challenges. [Online]. Available: https://www.ishn.com/articles/107192-food-safety-traceability-challenges

[4] ¿qué son los smart contracts o contratos inteligentes? [Online]. Available: https://www.miethereum.com/smart-contracts/

[5] Cadena de bloques. Page Version ID: 120668511. [Online]. Available: https://es.wikipedia.org/w/index.php?title=Cadena_de_bloques&oldid=120668511

[6] Home | ethereum. [Online]. Available: https://www.ethereum.org/

[7] Bitcoin - dinero p2p de código abierto. [Online]. Available: https://bitcoin.org/es/

[8] Sistema de prueba de trabajo. Page Version ID: 117642731. [Online]. Available: https://es.wikipedia.org/w/index.php?title=Sistema_de_prueba_de_trabajo&oldid=117642731

[9] Ledger — hyperledger-fabricdocs master documentation. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-1.4/ledger/ledger.html

[10] Blockchain privada vs. pública: ¿cual es la mayor diferencia? – latam. [Online]. Available: https://nemespanol.io/blockchain-privada-vs-publica-cual-es-la-mayor-diferencia/

[11] Smart contracts and chaincode — hyperledger-fabricdocs master documentation. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-1.4/smartcontract/smartcontract.html

[12] T. SL. Tokenización de inmuebles: factores a tener en cuenta. [Online]. Available: https://medium.com/@tokeniza/tokenizaci%C3%B3n-de-inmuebles-factores-a-tener-en-cuenta-aa9e1f6b1aa4

[13] BBVA. Qué es un 'token' y para qué sirve. [Online]. Available: https://www.bbva.com/es/que-es-un-token-y-para-que-sirve/

[14] Hyperledger indy – hyperledger. [Online]. Available: https://www.hyperledger.org/projects/hyperledger-indy

[15] Hyperledger composer - create business networks and blockchain applications quickly for hyperledger | hyperledger composer. [Online]. Available: https://hyperledger.github.io/composer/latest/

[16] What is a container? | docker. [Online]. Available: https://www.docker.com/resources/what-container

[17] Node.js. [Online]. Available: https://nodejs.org/es/

[18] {{ mustache }}. [Online]. Available: https://mustache.github.io/

[19] Express - infraestructura de aplicaciones web node.js. [Online]. Available: https://expressjs.com/es/

[20] Chai. [Online]. Available: https://www.chaijs.com/

[21] Mocha - the fun, simple, flexible JavaScript test framework. [Online]. Available: https://mochajs.org/

[22] node-dom - npm. [Online]. Available: https://www.npmjs.com/package/node-dom

[23] Angular - architecture overview. [Online]. Available: https://angular.io/guide/architecture

[24] ngx-toastr - npm. [Online]. Available: https://www.npmjs.com/package/ngx-toastr

[25] "Scrum_guide." [Online]. Available: https://www.scrumguides.org/docs/scrumguide/v1/scrum-guide-es.pdf

[26] Peers — hyperledger-fabricdocs master documentation. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-1.4/peers/peers.html

[27] The ordering service — hyperledger-fabricdocs master documentation. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-1.4/orderer/ordering_service.html

[28] Channel capabilities — hyperledger-fabricdocs master documentation. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-1.4/capabilities_concept.html

[29] Singleton. Page Version ID: 111190596. [Online]. Available: https://es.wikipedia.org/w/index.php?title=Singleton&oldid=111190596

)]noauthor$_u sabilidad_2 019 U sabilidad. Page Version ID$: 119179476.$[Online]. Available$:

Transferencia de estado representacional - wikipedia, la enciclopedia libre. [Online]. Available: https://es.wikipedia.org/wiki/Transferencia_de_Estado_Representacio