Escuela Técnica Superior de Ingeniería Informática

Grado en Ingeniería Informática Ingeniería del Software

TRABAJO FIN DE GRADO

NeptunUS

Digitalización de entornos portuarios con Blockchain y Smart Contracts

Autor/es:

Gonzalo García Granés

Tutor/es:

Dr. Ángel Jesús Varela Vaca

Dra. Antonia Mª Reina Quintero

Convocatoria de Junio

Curso 2020/21

Índice general

Índice de figuras

Índice de Tablas

Índice de Listados

Agradecimientos

Me gustaría dar las gracias a mis padres, por animarme con todo y apoyarme hasta el punto y final de la carrera, y más allá.

A todos los compañeros que he tenido durante estos años y en especial a Raquel, Manu, Belén, Ezequiel y Adrián por animarme tantas tardes de estudio.

A todo Anexo, por ser los mejores amigos que uno podría encontrar en un pueblo sueco perdido de la mano de Dios, y en especial a Nuria por animarme a usar LaTeX, para bien o para mal.

A Isaac, Anabel, Paula, María y Carla por sacar siempre un ratito para dar una vuelta, descansar y preocuparnos por tonterías casi todos los fines de semanas. Mi salud mental en estos cinco años habría ido a peor sin vosotros.

A Aura por, sin quererlo, ser mi mentora en esto de la tecnología, además de interesarse siempre en cualquier cosa que haga, y aguantar textos larguísimos explicándole cualquier pequeño pasito que para mí suponga un hito.

A Álex, Álvaro, Aritz y Jose Andrés por darme apoyo cada vez que el estrés se me echaba encima, aunque fuera telemáticamente.

Y especialmente, gracias a mis tutores, Ángel y Toñi, por animarse a hacer "algo de blockchain" conmigo y terminar ayudándome a ponerle cimientos, paredes y techo a este proyecto al que durante mucho tiempo no vi final. Sé que soy vuestro proveedor de spam favorito.

Resumen

Pese a que la sociedad y todos sus procesos están cada vez más digitalizados, los entornos portuarios se siguen viendo en una situación analógica y poco automatizada. Esta falta de digitalización, que debe suplirse con procesos manuales, supone problemas como la posibilidad de fallo humano, o sobrecostes tanto económicos como temporales. Esto es especialmente preocupante en una industria que supone el 20 % del PIB del sector del transporte en nuestro país.

De este modo, se ataca un problema concreto, la falta de digitalización de los entornos portuarios. Concretamente abordaremos el proceso de los registros de las escalas de los buques, pues es un proceso en el que apenas se han hecho avances en cuanto a la digitalización pese a ser uno por el que tiene que pasar todo buque que atraque en un puerto, independientemente del tipo de transporte que realicen. Sin embargo, intentaremos solucionar el problema utilizando, en lugar de cualquier otra tecnología, blockchain.

Al fin y al cabo, nos encontramos en pleno auge de las criptomonedas y, especialmente, de la tecnología que permite su funcionamiento. Las ideas de trazabilidad y descentralización son interesantes ya no solo desde un punto de vista económico, sino también para que los usuarios puedan confiar en una plataforma sabiendo de dónde vienen y hacia dónde van los datos, y que no están en un único lugar. Les aporta la seguridad almacenar los datos en varios nodos seguros en vez de un único servidor que puede fallar o ser modificado artificialmente. Viendo las posibilidades de la tecnología, surge la pregunta: ¿podríamos aprovechar una blockchain para solucionar la falta de digitalización del proceso de registro de las escalas aportando además trazabilidad, confiabilidad y descentralización?

Definida esta pregunta, hemos investigado cómo solucionar el problema utilizando blockchain como tecnología base para realizar el registro de datos. Esto nos ha llevado a introducirnos también en el uso de smart contracts. Tras un análisis de las alternativas para realizar la implementación, llegamos a una solución basada en HyperLedger Fabric como blockchain, Cicero como lenguaje de smart contracts y React como tecnología para una aplicación web. Se justifica además el uso de un gestor de procesos, en concreto Camunda BPMN, como estructura de unión entre estas tecnologías.

Finalmente, con una implementación en forma de prueba de concepto, ha quedado demostrado que el problema de la digitalización del registro de las escalas es solucionable usando blockchain como base. Sin embargo, hemos de tener en cuenta el rendimiento de la solución, que es peor de lo esperado, y reflexionar acerca de si las ventajas que tiene esta solución justifican su uso.

1 Introducción

Este capítulo servirá para adentrarnos en el proyecto, dando una visión general del problema que lo motiva así como del contexto en el que se desarrollará. Así, en el apartado 1.1, explicaremos con la motivación del proyecto, indicando qué problema queremos resolver. A continuación, en el apartado 1.2, contextualizaremos dicho problema y discutiremos brevemente las alternativas de implementación. Por último, en el apartado 1.3 definiremos los objetivos que pretenderemos alcanzar con este proyecto y en el apartado 1.4 comentaremos la estructura de este documento.

1.1 Motivación

Que la sociedad en la que vivimos se encuentra en un proceso de constante digitalización es, a estas alturas, una realidad tangible. Da igual la industria de la que hablemos, podemos ver cómo se han ido moviendo hacia un modelo de negocio en el que los sistemas digitales no son ya un apoyo sino la base de sus negocios. Sin embargo, este proceso de digitalización no se ha producido con la misma velocidad ni éxito en todos los ámbitos industriales. Y, en los entornos portuarios, nos encontramos con una situación especialmente insuficiente.

Hablamos de una industria, la portuaria, que, pese a suponer el 20 % del PIB del sector transportes en España, empieza ahora a digitalizar sus infraestructuras y procesos [18, 17]. Así, procesos que podrían estar automatizados se convierten en tareas manuales y tediosas. Más allá del problema de aumento de costes que esto puede producir, tanto temporales como económicos, la falta de digitalización supone una peor experiencia para los usuarios, además de introducir en el proceso la probabilidad de fallo humano. Se empiezan a hacer avances, por ejemplo, en el control de contenedores [27], pero hay un proceso que se encuentra todavía en una situación muy precaria en cuanto a su digitalización. Hablamos del proceso de registro

de escalas, uno por el que debe pasar cualquier tipo de buque, ya sea de transporte de personas, vehículos, contenedores... Nos centraremos por tanto en este problema concreto, la falta de digitalización del registro de escalas, e intentaremos solventarlo usando una de las tendencias de la llamada *industria 4.0*: blockchain [19].

La tecnología **blockchain** se ha convertido, en los últimos tiempos, en una de las más populares [22]. Su papel como soporte tecnológico para las criptomonedas (Bitcoin [40], Ether [29], Dogecoin [32], etc.) la ha convertido en un término que poco a poco se afianza en el vocabulario popular, tal y como lo hicieran en su día los términos "Internet" o "base de datos". Sin embargo, es popularmente conocido por tener un carácter enfocado a lo económico. Por ejemplo, en los medios de comunicación, si no se utiliza en relación a estas monedas, se hace al hablar de los llamados Tokens No Fungibles (NFTs) [23], que también se utilizan para intercambiar y acumular riqueza. Sin embargo, nos podemos plantear la pregunta de si puede tener esta tecnología otros usos.

Sin embargo, es evidente que es una tecnología con mucho potencial y que probablemente tiene usos más allá del meramente económico. No en vano, aporta un enfoque distinto en cuanto al registro de datos frente a las bases de datos popularizadas hasta ahora. Aun así, como veremos en más profundidad en el apartado 1.2.3, blockchain se define como un "libro de contabilidad". Tal vez, partiendo de esa definición, pueda parecer difícil darle un uso más centrado en los datos.

No podemos olvidar, sin embargo, que blockchain como tecnología se basa en los conceptos de descentralización y trazabailidad. Esto quiere decir que cualquier registro que aparezca en la blockchain no se almacena en un único lugar sino que se reparte entre varios puntos o nodos, y que podemos tener datos (ya sean públicos o bien anonimizados) sobre quién realizó el registro y de cuándo se hizo. Además, estos registros son inmutables, lo que nos añade una capa de confianza, ya que si los datos son verificados y se añaden a la blockchain, podremos confiar en ellos ya que no podrán ser modificados. Todo esto es interesante para un "libro de contabilidad", por supuesto, pero también para cualquier otro registro de datos. Así nos surge la pregunta alrededor de la que versará este proyecto: **¿es factible usar blockchain como tecnología alternativa para el registro de datos?**.

En resumen, el problema último que busca resolver este proyecto es la transformación digital de los procesos en entornos portuarios. Concretamente, tomaremos el proceso de registro de escalas y, para especificarlo aún más, lo analizaremos en el puerto de Santander. Para unir dicha problemática a la transformación hacia la industria 4.0, además de a la pregunta que motiva la realización del proyecto, intentaremos hallar una solución que involucre la tecnología blockchain, y de este modo aprovechar sus ventajas en un caso que no se relaciona directamente con lo económico.

Aunque el proyecto será realmente una prueba de concepto y no tanto una aplicación final, nos enfrentaremos al reto de forzarnos a realizar esta digitalización con una solución basada en blockchain, y que al mismo tiempo aporte comodidad a los futuros usuarios. Una vez demostremos que es realizable, este proyecto nos debe servir para analizar hasta qué punto es factible, ya no desde el punto de la tecnología sino desde los costes que genera.

1.2 Contexto

Partiendo del problema específico de digitalizar el sistema de registro de escalas en el puerto de Santander utilizando la tecnología blockchain, hay una serie de conceptos que el lector de este documento debe de conocer: la propia tecnología blockchain, pero también qué son los smart contracts y un gestor de procesos. Al final, en la implementación que haremos en este proyecto, no sólo usaremos blockchain como base sino que necesitaremos los smart contracts para realizar transacciones y el gestor de procesos para unir todas las tecnologías sirviendo como *middleware*. Será por tanto el objetivo de esta sección introducir estas tres tecnologías.

1.2.1 Sobre los smart contracts

Los *smart contracts* son un concepto introducido por Nick Szabo, que los define como "protocolos con interfaces de usuario para formalizar y asegurar relaciones sobre redes informáticas" [46]. En el caso de blockchain, los podemos definir como contratos *autoejecutables* [28]. Esto se refiere a que el contrato, desplegado en una blockchain, se encarga de comprobar qué condiciones se cumplen y lleva a cabo las obligaciones asociadas. Por ejemplo, realizar una transferencia si un usuario sigue suscrito a un servicio en determinada fecha. Tienen que ser, por tanto, capaces de realizar el proceso de verificación sin involucrar a terceros, sin intervención humana. Y esto nos permitirá automatizar acciones, algo a lo que sacaremos provecho en este proyecto.

Por lo general, cada plataforma de blockchain tiene su propio lenguaje para smart contracts. Por ejemplo, Solidity en Ethereum o Script en Bitcoin. Podemos ver un ejemplo de este último en el listado 1.1. Sin entrar en detalle en la complicada sintaxis de Script, este ejemplo es una transacción de una cierta cantidad a una clave pública.

En este proyecto usaremos Cicero, un lenguaje con una sintaxis mucho más sencilla, similar al lenguaje natural, lo cual facilitará la modificación de los contratos por parte del usuario encargado de mantenerlo al día con respecto a lo que indique la ley en un momento concreto. Hablaremos con más detalle de este lenguaje, desarrollado por Accord Project, en el apartado 5.2.

Listado 1.1: Ejemplo de smart contract en Bitcoin Script

```
1 scriptPubKey: OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG
2 scriptSig: <sig> <pubKey>
```

1.2.2 Sobre el gestor de procesos

Esta es, tal vez, la parte cuya necesidad se entiende de manera menos intuitiva. Sin embargo, el gestor de procesos terminará siendo el pilar en torno al cual girará toda la estructura de la implementación. No en vano, un gestor de procesos nos permite desplegar y automatizar procesos, los cuales usaremos para orquestar las distintas actividades de las

que se compone el flujo de la aplicación, ya sean humanas (la introducción de datos por un usuario) o automáticas (las llamadas a la blockchain).

Al gestor de procesos le sacaremos partido de dos formas: por un lado, lo usaremos para convertir los *inputs* del usuario en datos que la blockchain sea capaz de entender y procesar; por otro, nos permitirá asegurar que el sistema funciona de manera secuencial, definiendo explícitamente que procesos deben ejecutarse antes que el resto. Esto, que en principio parece evidente (al final, cualquier aplicación es un conjunto de procesos que se ejecutan en orden concreto, sin necesidad de un gestor de procesos que los ordene) será fundamental en este proyecto, debido a la diversidad de tecnologías utilizaremos. Analizaremos con más detalle el uso del gestor de procesos en el apartado 5.4.

1.2.3 Sobre blockchain

Blockchain se define como un *ledger* o "libro de contabilidad" compartido e inmutable que facilita el proceso de registro de las transacciones y el seguimiento de los activos en una red empresarial [33]. Supongamos que trabajamos en un conjunto de equipos (llamémoslos nodos, servidores, personas, etc.) conformando una red en la que suceden intercambios de información, es decir, transacciones. Un *ledger* público permitirá que todo aquel que pertenezca a la red pueda saber, con más o menos detalle, qué está ocurriendo en ella.

Podemos ver un esquema conceptual de la arquitectura general de una blockchain [37] en la Figura 1.1. Vemos cómo, después de que un usuario realice una transacción, esta se emite como bloque al resto de equipos o nodos de la red. Una vez que dichos nodos comprueban y aprueban la transacción, esta se añade a la blockchain como tal, aparece registrada en el libro de contabilidad o *ledger*. Una vez añadida a la blockchain, la transacción se vuelve inmutable, no puede ser modificada [45]. Es por ello que la comprobación, que se realiza de manera automática antes de que la transacción sea inmutable, añade una capa de confianza: si la transacción es correcta, será siempre correcta, podremos confiar siempre en ella [47].

Uno de los mejores ejemplos es el del banco, que de nuevo nos sitúa en el mundo de la economía, pero que es al mismo tiempo su uso más popular. Típicamente, todos los "libros de contabilidad" que usamos están guardados en uno o unos pocos lugares, los bancos. Podemos, por supuesto, acceder a una copia de nuestras cuentas, saber cuánto dinero tenemos, cuánto y cuándo lo hemos gastado y un medio para subsanar posibles errores en esas cuentas. Sin embargo, a la hora de intercambiar dinero con un tercero, tenemos que confiar del banco para dicha comprobación. Y además, si se diera el caso de que el banco perdiera ese libro (y no tuviera copias de seguridad), toda esa información desaparecería para siempre. Esto último es algo que, desde la ficción, ya han explorado series como *Black Mirror* o *Mr Robot* [35], lo que demuestra que es cada vez más una preocupación real de la sociedad.

Blockchain es además aplicable en otros contextos, aunque sean algo menos comunes. Algunos ejemplos son [25]:

- **Notaría**, como registro de la existencia de documentos. Se podría saber en qué momento se supo de la existencia del documento y que contenía. Podría usarse para

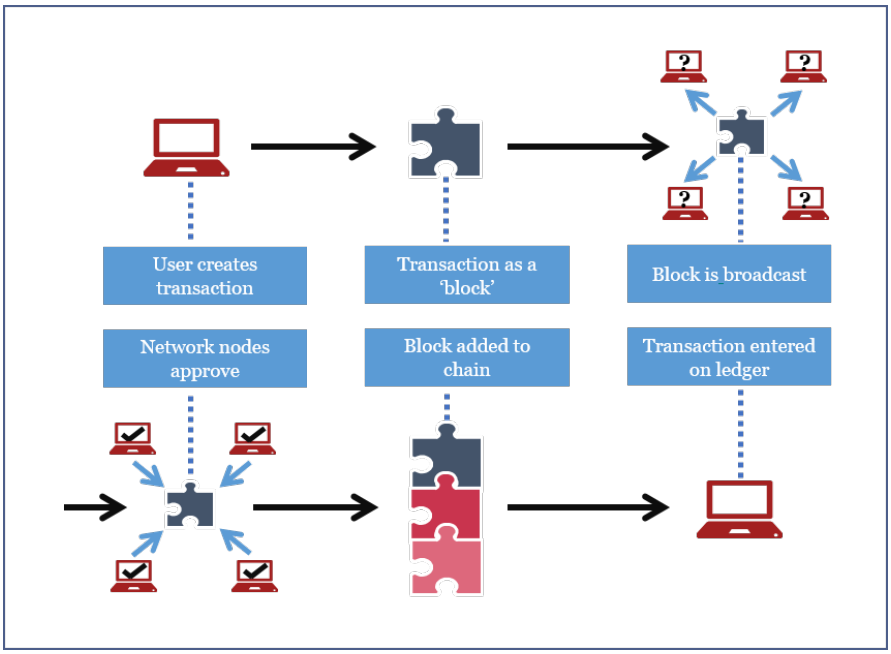

Figura 1.1: Gráfico conceptual de la arquitectura de una blockchain [37]

evitar la destrucción de documentos (el registro nunca se podría borrar) así como la falsedad documental (habría disponible una copia inmutable del documento). Un ejemplo de este uso es la aplicación *Stampery* [24].

- **Industrial**, como en el caso de la compañía danesa de contenedores de mercancías *Maersk*. Esta empresa desarrolló junto a IBM un sistema basado en blockchain para mantener toda la información acerca del rastreo y estado de sus contenedores [27].
- **Industria musical**, para registrar qué parte de las royalties (el dinero que se gana por cada reproducción) pertenece a cada uno de los actores (artista, discográfica, sello, mánager, etc.) Así se podrían automatizar las transacciones y además habría un registro confiable sobre ciertas condiciones del contrato (por ejemplo, discográficas que aceptan cobrar menos porcentaje de royalties cada año, o posibles rupturas de cláusulas del contrato).
- **Almacenamiento descentralizado**, como alternativa a soluciones como *Dropbox* o *Google Drive* que nos obligan a confiar en un tercero para almacenar nuestra información. Plataformas como *Storj* [49] utilizan el excedente en espacio y ancho de banda de sus usuarios (que reciben a cambio pagos en Bitcoin) para almacenar archivos encriptados cuya integridad y disponibilidad se comprueban periódicamente contra lo registrado en la blockchain.

En concreto, para nuestro proyecto se ha usado HyperLedger, una blockchain desarrollada principalmente por IBM. La usaremos por dos motivos principales: está pensada para un

uso más empresarial (lo cual da pie a desarrollar aplicaciones que no necesitan el soporte de una criptomoneda [30]) y Cicero, el lenguaje de smart contracts que usaremos, está especialmente bien integrado con esta tecnología. Discutiremos más en profundidad las ventajas y la implementación de esta tecnología en el apartado 5.3.

1.3 Conclusión y objetivos

Como hemos explicado a lo largo de este capítulo, el proyecto tiene una doble problemática. Una más general y abstracta, que será demostrar que podemos usar blockchain para el registro de datos más allá del ámbito puramente económico. Y otra más concreta y funcional, que será digitalizar el sistema de registro de escalas de buques en el puerto de Santander. Por tanto, el proyecto tendrá tres objetivos claros:

1. Investigar e introducirnos en el uso de tecnologías de blockchain y smart contracts.
2. Implementar una infraestructura usando blockchain y smart contracts que nos permita registrar datos.
3. Implementar un caso de uso concreto, usando dicha infraestructura para el registro de escalas que realizan los buques en el puerto de Santander.

Los objetivos siguen, al final, la secuencia que va desde la introducción en una nueva tecnología hasta su uso real. Es destacable, eso sí, que el objetivo último del proyecto no es terminar con una aplicación que un usuario final pueda usar. Nuestro trabajo servirá más bien para analizar la viabilidad de la solución y dejar el camino marcado para una futura implementación real.

1.4 Estructura de la memoria

Los capítulos que conforman esta memoria siguen la siguiente estructura:

- En el capítulo 2, detallaremos la planificación que se ha seguido para el proyecto y estimaremos tanto el coste temporal como el económico del mismo.
- En el capítulo 3, elicitaremos los requisitos de la aplicación, a partir del análisis del modelo de negocio actual del puerto de Santander.
- En el capítulo 4, realizaremos un definiremos los casos de uso y el modelo de datos de la aplicación, además de discutir sobre las alternativas que teníamos para la arquitectura de la aplicación.
- En el capítulo 5, analizaremos la arquitectura final de la aplicación e indagaremos sobre la implementación y las diversas tecnologías utilizadas.

- Por último, en el capítulo 6, reflexionaremos sobre lo conseguido en el proyecto y la viabilidad de la idea en el mundo real.

Se incluyen además dos anexos: el Anexo A incluye un manual de despliegue de la aplicación; mientras que el Anexo B detalla cómo usar la aplicación a nivel de usuario.

2
Planificación y costes

Este capítulo se dedicará a detallar la estimación en tiempo, además de cuánto se dedico a las diversas tareas y un breve análisis sobre los desvíos en la planificación. También estimaremos los costes del proyecto. Así, en la sección 2.1 nos centraremos exclusivamente en el tiempo, comentando qué tareas se definieron para la realización del proyecto, en cuánto tiempo se estimaron y cuánto se tardó en completarlas. En la sección 2.2, comentaremos el coste económico que estimamos para la realización de este proyecto, y en la sección 2.3 concluiremos el capítulo.

2.1 Estimación temporal

Tanto para la estimación del tiempo como para controlar el tiempo que finalmente se ha empleado en el proyecto, se utilizó un sistema basado en Redmine [12], concretamente Gestión de Proyectos [8], aplicación para la creación y gestión de tareas proporcionada por la Universidad de Sevilla. El tiempo necesario para llevar a cabo el proyecto se estimó en **306 horas** y se realizó en 249 horas. En las tablas 2.1, 2.2, 2.3 y 2.4 aparece el desglose de estimación y tiempo real de cada tarea según su tipo: investigación, programación, organización y escritura, respectivamente. Se ha intentado que las tareas fueran lo más atómicas posibles, de manera que en el propio nombre quedara claro cuál era el criterio para decidir que la tarea estaba completada. Esto permite, por un lado, estimar más claramente (no estimamos cuánto se tardará en hacer la implementación, sino tareas mucho más concretas), y por otro, tener una visión clara de qué hemos hecho y qué queda por hacer.

Fijándonos ahora en la tabla 2.5, que es un resumen general de la estimación y el tiempo real según el tipo de tarea, podemos ver que hemos hecho una sobrestimación del tiempo. Salvo en contadas excepciones en las que algo ha complicado de más la tarea; por ejemplo,

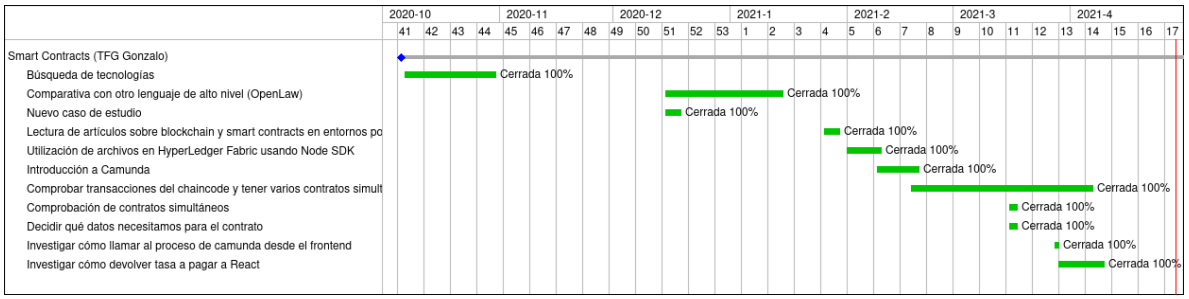

Figura 2.1: Diagrama de Gantt de las tareas de investigación

la familiarización con Docker, que conllevó desinstalar y reinstalar tres veces el software por problemas de versiones, siempre se han podido completar las tareas en torno a una hora antes de lo estimado. La diferencia total entre lo estimado y lo trabajado es de 57 horas. De todas formas, hay que considerar que era la primera vez que nos enfrentábamos a un proyecto de este calibre, así que cierto error en la estimación es asumible. De cara a futuros proyectos tendremos más experiencia sobre cuánto tiempo nos puede llevar realizar las tareas.

A continuación, podemos ver unos diagramas de Gantt, generados por la plataforma Gestión de Proyectos. Así, la figura 2.1 se corresponde a las tareas de investigación; la figura 2.2, a las tareas de programación; la figura 2.3, a las de organización y la figura 2.4, a las de escritura. Estos diagramas nos permiten visualizar cómo se han repartido las tareas a lo largo del tiempo que hemos empleado en la realización del proyecto, separadas de nuevo por tipo. Finalmente, la figura 2.5 nos permite ver cómo se han realizado todas las tareas del proyecto.

En estos diagramas, especialmente en la 2.5, podemos comprobar que este proyecto ha sido bastante dilatado en el tiempo. Las primeras tareas de investigación, así como las de familiarización con las tecnologías se realizaron entre octubre y diciembre de 2020, mientras que el grueso del proyecto se realiza a partir de enero. Conforme nos acercamos a la finalización de la implementación (mes de abril) las tareas dejan de hacerse secuencialmente para pasar a trabajar más en paralelo. Esto se debe a que, a partir de cierto punto en la implementación, las tareas no son tan atómicas. Por ejemplo, mientras se termina la validación del formulario también se está comprobando que las llamadas a la API son correctas. Las dos últimas tareas que son la realización de esta memoria y su mejora, se realizan aisladas, ya que decidimos dejarlo hasta que tuviéramos el conocimiento completo del abarque del proyecto y de cómo se había realizado la implementación.

2.2 Análisis de costes

Comenzando con el coste del personal, teniendo en cuenta que el sueldo (bruto) medio de un perfil junior (como el del autor de este proyecto) es de aproximadamente 21.000€ [20], podemos estimar en 11€/h el coste por persona. Dado que sólo ha participado una

Tareas de investigación	Tiempo estimado	Tiempo real
Búsqueda de tecnologías	15:00	12:00
Comparativa con otro lenguaje de alto nivel (OpenLaw)	8:00	7:00
Nuevo caso de estudio	9:00	7:00
Lectura de artículos sobre blockchain y smart contracts en entornos portuarios	8:00	8:00
Utilización de archivos en HyperLedger Fabric usando Node SDK	8:30	6:00
Introducción a Camunda	13:00	9:00
Comprobar transacciones del chaincode y tener varios contratos simultáneos	9:00	10:00
Comprobación de contratos simultáneos	3:30	2:40
Decidir qué datos necesitamos para el contrato	2:00	1:30
Investigar cómo llamar al proceso de Camunda desde el frontend	2:00	1:45
Investigar cómo devolver tasa a pagar a React	8:00	6:00
	86:00	**70:55**

Tabla 2.1: Planificación temporal de las tareas de investigación

Tareas de programación	Tiempo estimado	Tiempo real
Creación de primera versión del contrato del puerto de Santander	10:00	10:00
Desplegar contrato en entorno de pruebas	20:00	15:20
Crear contrato caso de uso de Brasil	5:00	3:00
Familiarización con Docker	25:00	29:00
Instalación de Ubuntu	6:00	5:30
Mejora script de despliegue del contrato	5:00	3:20
Hacer proceso en Camunda que interactúe con el contrato	10:00	10:30
Crear contrato con todos los artículos	8:00	7:00
Hacer log de IALA	5:00	4:40
Crear formulario de petición	8:00	6:00
Hacer nuevo worker que cree JSON de request	5:00	5:15
Terminar validaciones frontend	6:00	6:00
Crear nuevo modelo de proceso	7:00	6:30
Mejoras finales	5:00	4:30
	125:00	**116:35**

Tabla 2.2: Planificación temporal de las tareas de programación

Tareas de organización	Tiempo estimado	Tiempo real
Puesta al día Gestión de Proyectos	5:00	3:30
Creación índice preliminar	3:00	1:30
Organizar archivos del proyecto	3:00	3:30
	11:00	**8:30**

Tabla 2.3: Planificación temporal de las tareas de organización

Tareas de escritura	Tiempo estimado	Tiempo real
Documentación contrato	4:00	3:45
Creación de primer borrador de la memoria	40:00	28:50
Mejora de la memoria	40:00	20:25
	84:00	**53:00**

Tabla 2.4: Planificación temporal de las tareas de escritura

Tipo de tarea	Tiempo estimado	Tiempo real
Investigación	86:00	70:55
Programación	125:00	116:35
Organización	11:00	8:30
Escritura	84:00	53:00
	306:00	**249:00**

Tabla 2.5: Comparación entre el total estimado y el real según el tipo de tarea

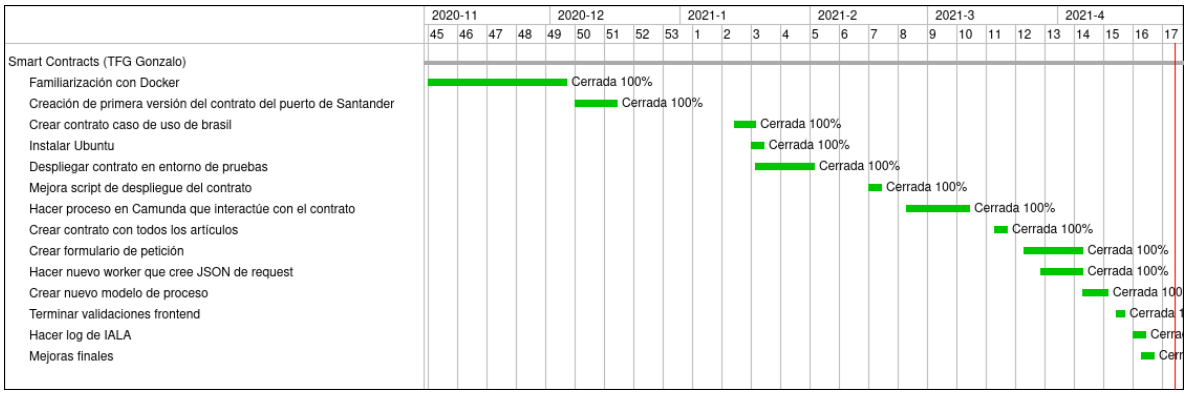

Figura 2.2: Diagrama de Gantt de las tareas de programación

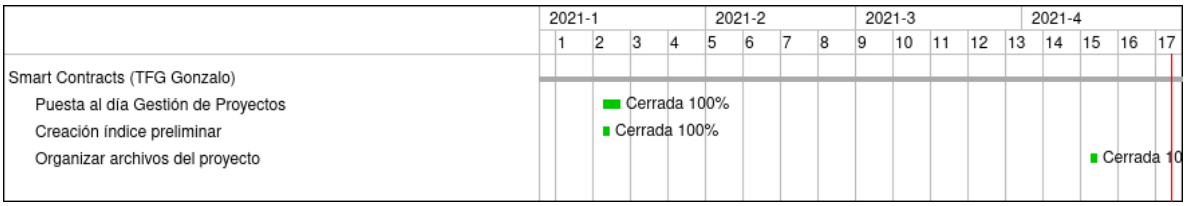

Figura 2.3: Diagrama de Gantt de las tareas de organización

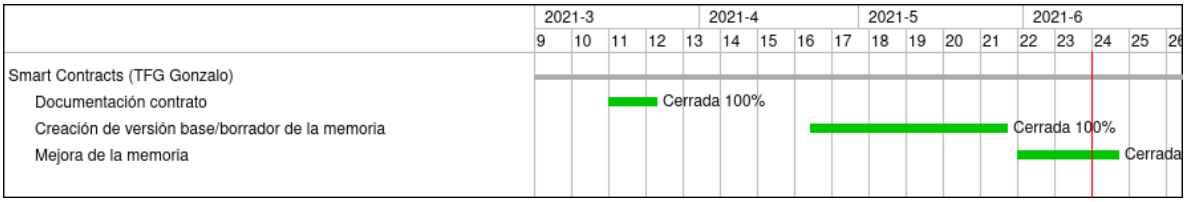

Figura 2.4: Diagrama de Gantt de las tareas de escritura

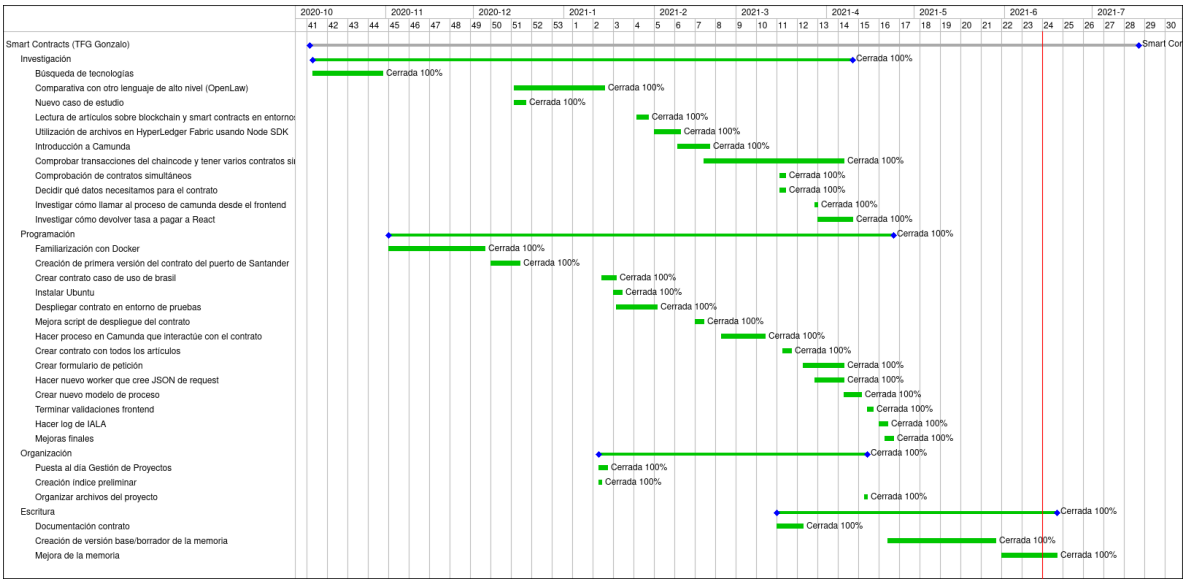

Figura 2.5: Diagrama de Gantt de todo el proyecto

persona en el proyecto, y que según la sección 2.1 se han dedicado 249 horas al proyecto, el coste del personal será de **2.739€**.

Respecto al hardware, en la tabla 2.6 se encuentran desglosados los costes en del equipo utilizado en la realización de este proyecto. Con respecto al software, todo lo que se ha utilizado ha sido o bien open source o bien en su versión gratuita. Aún así, en la tabla 2.7 aparece el desglose, también para que quede como listado de todo el software necesario en el proyecto.

Por tanto, el coste total de este proyecto, incluidos impuestos, asciende a **4.168,92€**.

Equipo	**Coste**
Ordenador portátil MSI PS42 8RB	1.149,99€
Monitor BenQ GL2460	124,95€
Ratón Logitech M185	19,99€
Auriculares Audio Technica ATH-MSR7B	134,99€
Coste total	**1.429,92€**

Tabla 2.6: Coste de los suministros de hardware utilizados en el proyecto

Software	**Coste**
Ubuntu 20.04.1 LTS	0,00€
Camunda Open Source Community Edition	0,00€
Camunda Open Source Community Modeler	0,00€
Docker Desktop Free Tier	0,00€
HyperLedger Fabric	0,00€
Herramientas de Accord Project	0,00€
Editor de LaTeXOverleaf Free Tier	0,00€
Coste total	**0,00€**

Tabla 2.7: Coste del software utilizado en el proyecto

2.2.1 Sobre el coste de la aplicación real

Estimar el coste real que supondría mantener este proyecto si llegara a ser una aplicación final es bastante complejo desde la situación actual, pues dependería de muchas variables. Por ejemplo, el coste es muy distinto si usamos una plataforma en la nube (como Azure de Microsoft o Amazon Web Services) que si pretendemos desarrollar nuestro propio sistema de nodos. También es difícil estimar el tamaño que sería necesario ya que, sin hablar directamente con el cliente, no podemos tener una idea real de la escala que debería tener la implementación.

Sin embargo, a efectos de completitud de esta documentación, vamos a hacer una estimación muy superficial del coste que supondría mantener esta blockchain. Vamos a usar

para ello los costes definidos para la versión preliminar del Blockchain Service de Microsoft Azure [21], ya que sería una alternativa posible para alojar nuestro sistema. Quedaría a expensas de una futura implementación el investigar si es la mejor plataforma para nuestras necesidades.

Primero, tenemos que aclarar que cuando Azure se refiere a *miembro* no se refiere a un usuario de la aplicación, sino al punto de entrada con el que los usuarios contactan para hacer una transacción. Así, vamos a considerar que tendremos 2 miembros, uno para cada zona que define el puerto de Santander. Del mismo modo, vamos a considerar que cada miembro tiene 3 nodos, una cantidad que nos asegura la descentralización pero que puede ser correcta para el volumen de transacciones diarias que se pueden realizar en cada una de las zonas. Por último, vamos a considerar que cada nodo tiene una capacidad de almacenamiento de 250GB. Esta cantidad es la que aparece recomendada por la guía de precios de Azure. Por supuesto, todos estos números habría que revisarlos de cara a hacer una implementación real.

Usando las formulas que aparecen en la web de Azure, mensualmente tendremos un coste de los nodos de 1.174,716€ y un coste de almacenamiento de 64,5€. Por tanto, podemos estimar que el coste mensual de desplegar una blockchain similar a la que creemos que necesitaría este proyecto es de **1.239,216€**.

2.3 Conclusiones

En este capítulo, hemos indicado, por una parte, la planificación que se hizo del trabajo (y sus diferencias con la realidad) y, por otra, los costes del proyecto. Debido a la naturaleza de este proyecto como prueba de concepto, es cierto que los costes son mucho más reducidos que los que supondría realizar una implementación real.

De hecho, como hemos visto en el apartado 2.2.1, el coste mensual de lo que suponemos que se necesitaría para una aplicación final casi equivalente al coste de todo el equipo que se ha usado en este proyecto. No obstante, la implementación real no tendría una investigación tan alargada en el tiempo, algo que reduciría costes de personal. En conclusión, por tanto, hay que entender este capítulo como concreto para este proyecto, y que difícilmente se podrían escalar planificación y estimación de costes a la realización de una aplicación final.

3
Elicitación del problema

Este capítulo comenzará detallando, en la sección 3.1, cuál exactamente es la situación actual del problema que pretendemos resolver. En la sección 3.2 analizaremos cómo se lleva a cabo ahora mismo el registro de una escala, comprobaremos su modelo de negocio e identificaremos los actores que participan en él. Terminaremos elicitando los requisitos concretos que debe cumplir esta implementación en la sección 3.3, y concluiremos el capítulo con la sección 3.4.

3.1 Dominio del problema y sistema actual

Tras haber investigado la situación actual en cuanto a la digitalización de los sistemas portuarios, hemos determinado que, en general, se encuentran en un estado bastante primitivo. Y, en cuanto a la manera en que se registran las escalas de un buque, aún más. Actualmente, el método es prácticamente manual, con apenas ningún automatismo. Así, cuando un agente marítimo quiere solicitar una escala, la autoridad portuaria debe registrar la solicitud de llegada, informar de dicho registro, esperar a que el agente marítimo genere la solicitud de los servicios que usará en el puerto, registrar esta segunda solicitud y entonces comunicar la previsión de escalas y la disponibilidad de los recursos.

Este sistema tiene a su favor, ante todo, que es el que se utiliza en todos los puertos de España y que, por tanto, todos los agentes implicados ya conocen a la perfección cómo funciona. Además, debido a lo manual que es todo, no supone un coste tecnológico elevado, ya que con una conexión normal a Internet se pueden realizar todas estas comunicaciones. De hecho, en general se realizan vía correo electrónico.

Sin embargo, tiene dos puntos en contra igualmente importantes. El primero es la enorme pérdida de tiempo que produce toda la burocracia que hay en el proceso. Por ejemplo, para

saber si hay escala disponible o no, hay que hacer dos solicitudes. Además, en todas estas solicitudes y documentos que se mandan entre agentes, se puede llegar a perder la veracidad de los datos: no hay una manera clara de asegurar que los datos no han sido modificados entre envíos.

Es por ello que proponemos una solución mucho más automatizada basada en una blockchain. En el caso del puerto, habilitaremos un sistema que permita a cualquier buque que quiera realizar una escala el poder registrar su petición, así como automatizar la respuesta a dicha petición centralizando todas las peticiones sobre un smart contract y accesible desde la blockchain.

3.2 Modelo de negocio y actores

En la figura 3.1 podemos ver parte del modelo de negocio actual, que hemos obtenido del sistema con el que funciona actualmente el puerto de Santander. Señalado en rojo podemos ver la parte del modelo en la que nos centraremos en este proyecto. A partir del modelo de negocio podemos identificar tres actores:

- **Agente marítimo - buque**: se trata de la persona que realiza la petición de escala en nombre del buque. En la figura 3.1, corresponde a la segunda pool (o línea) del diagrama. Cuando se prepara la llegada del buque al puerto, el agente marítimo informa a la autoridad portuaria de que quiere realizar una escala. Una vez que recibe confirmación de que su solicitud se ha registrado, genera una solicitud de los servicios de los que quiere hacer uso en la escala y la emite a la autoridad portuaria. Una vez que el práctico recibe la previsión de escala y la disponibilidad de recursos, el buque realiza su aproximación y, una vez que llega a su puesto, comunica a la Autoridad portuaria de su llegada.

- **Autoridad portuaria**: es la entidad (una o varias personas) que registran las peticiones del buque que solicita la escala y que responden de acuerdo a la situación actual. En la figura 3.1, corresponde a la pool superior. Cuando la autoridad portuaria recibe una solicitud de escala del agente marítimo de un buque, la registra e informa de dicho registro al agente marítimo. A continuación, recibe la solicitud de servicios, que registra, e informa de nuevo al agente marítimo de la previsión de escalas y la disponibilidad de recursos, de acuerdo con lo que haya solicitado. Por último, espera hasta que recibe el comunicado de que, efectivamente, el buque ha llegado al puerto.

- **Práctico**: es la persona intermediaria entre el buque y la autoridad marítima, que conoce el puerto y se encarga de dirigir el buque hacia el puesto asignado. En la figura 3.1, se representa en la pool inferior. Una vez que la autoridad portuaria comunica la previsión de escalas y la disponibilidad de recursos, se encarga de realizar la aproximación. Por último, cuando el buque comunica que ha llegado a su puesto, recibe la notificación de la llegada por parte de la Autoridad portuaria. Lo hemos incluido

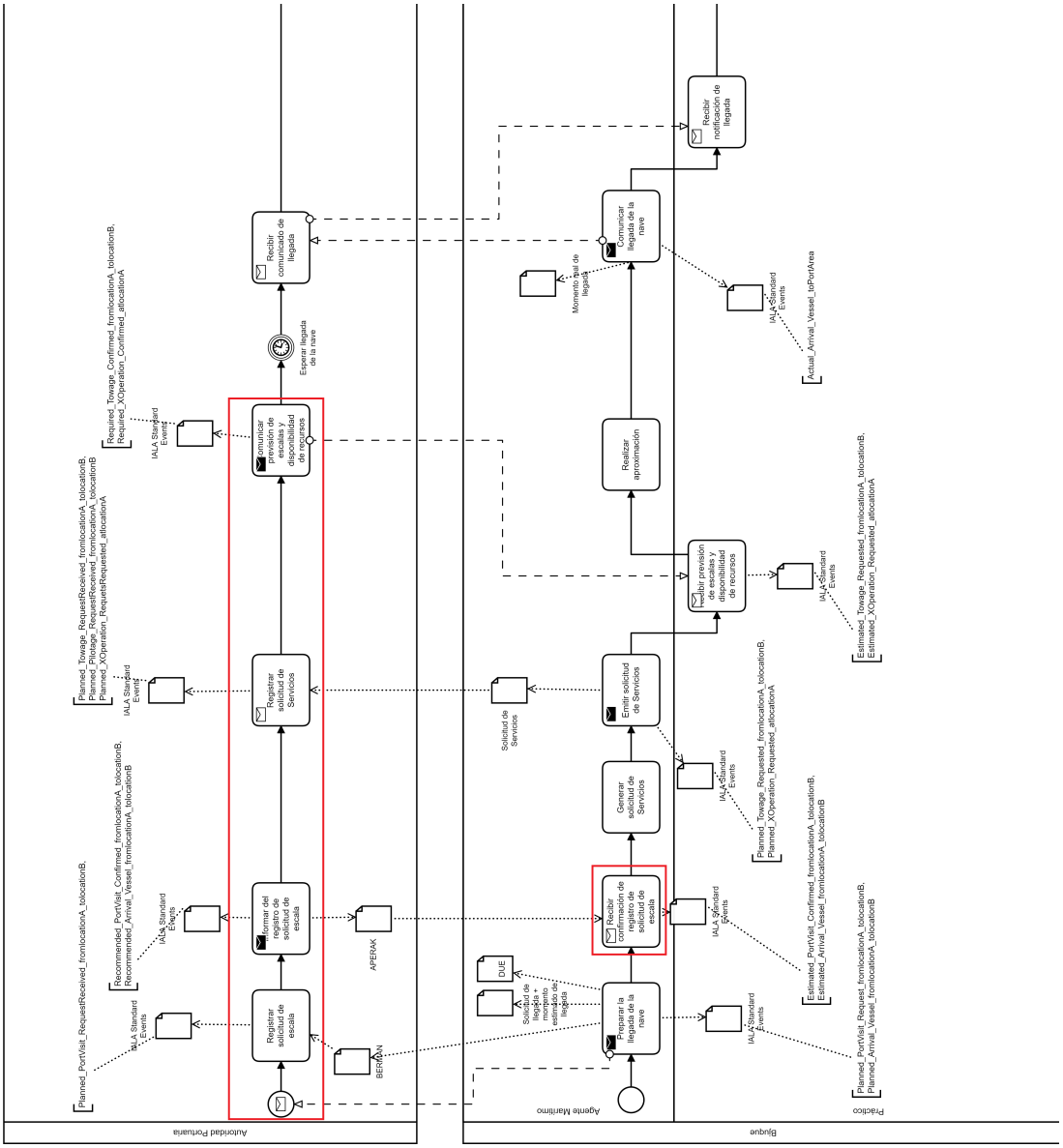

Figura 3.1: Vista del modelo de negocio actual [34].

ya que aparece siempre en toda la documentación del modelo de negocio actual del puerto de Santander. Sin embargo, todo el proyecto se hará desde el punto de vista exclusivamente de la comunicación entre autoridad portuaria y el agente marítimo, ya que al práctico le resulta indiferente el cómo obtener dónde tiene que atracar el buque.

3.3 Requisitos de la solución

A continuación, especificaremos los requisitos que debe cumplir la solución que vamos a desarrollar. Comenzaremos con los requisitos funcionales, con los que describiremos cómo esperamos que la aplicación se comporte. Los requisitos no funcionales nos permitirán describir expectativas más abstractas, que no se relacionan con una característica concreta de la aplicación. Por último, junto a los requisitos de información, definiremos algunos términos con los que nos hemos familiarizado en la realización de este proyecto.

3.3.1 Requisitos funcionales

A continuación, se describen los requisitos funcionales de la solución que planteamos. Estos requisitos funcionales aparecen en la tabla 3.1, indicando para cada uno de ellos un identificador, un título, una descripción breve y el actor o actores que se benefician de la implementación del requisito.

ID	Título	Descripción	Actor
RF-01	Registro de escala	El sistema debe permitir registrar una escala a un buque en una de las zonas del puerto habilitadas para ello.	Agente marítimo
RF-02	Cálculo de tasa	El sistema debe calcular automáticamente el coste de la tasa que tendrá que pagar el buque de acuerdo al contrato de la escala.	Agente marítimo
RF-03	Actualización del contrato	El sistema debe permitir modificar el contrato a partir del cual se calculan las tasas en caso de que la ley sufra alguna modificación.	Autoridad portuaria
RF-04	Información de registro	El sistema debe informar de si el registro se ha podido realizar, si no quedan escalas disponibles o si ha ocurrido algún otro problema.	Agente marítimo
RF-05	Logs de IALA	El sistema debe generar los logs necesarios según IALA (International Association of Lighthouse Authorities).	Autoridad portuaria Agente marítimo

Tabla 3.1: Requisitos funcionales

Con respecto al requisito RF-05, debido a la complejidad de los logs necesarios para cumplir con la normativa IALA, que se salen del ámbito del proyecto, en esta prueba de concepto demostraremos que se pueden emitir pero no lo haremos con la sintaxis y tipo de archivos correctos. Utilizaremos una versión más manejable de esta sintaxis, que nos sirve para los propósitos del proyecto. De este modo, tendremos tres tipos de eventos IALA que aparecerán en los logs:

- `IALA_SCALE_AVAILABLE_{MMSI}`: Se ha encontrado escala disponible.
- `IALA_SCALE_UNAVAILABLE_{MMSI}`: No se ha encontrado escala disponible.
- `IALA_TAX_TO_PAY_{MMSI}`: Si hay escala disponible, cuánto hay que pagar de tasa.

Podemos encontrar una muestra de estos logs en el listado 3.1.

Listado 3.1: Ejemplo de logs con nuestra implementación de la sintaxis IALA

```
1 [2021-04-15T17:10:25.031Z] IALA_SCALE_AVAILABLE_353136000: Scale
    available for ship EVER GIVEN
2 [2021-04-15T17:10:25.444Z] IALA_SCALE_UNAVAILABLE_9383936: Scale not
    available for ship OASIS OF THE SEAS
3 [2021-04-15T17:10:29.773Z] IALA_TAX_TO_PAY_353136000: The amount to pay
     for the ship EVER GIVEN is 9732.47\euro
4 [2021-04-15T17:10:44.575Z] IALA_SCALE_AVAILABLE_224072000: Scale
    available for ship SEVILLA KNUTSEN
5 [2021-04-15T17:10:49.643Z] IALA_TAX_TO_PAY_224072000: The amount to pay
     for the ship SEVILLA KNUTSEN is 10103.54\euro
```

3.3.2 Requisitos no funcionales

El principal requisito no funcional de esta aplicación es, por el propio diseño del proyecto, que tiene que usar blockchain para el registro de las escalas. Este y el resto de requisitos no funcionales aparecen en la tabla 3.2. En este caso, sólo indicamos identificador, título y descripción de cada requisito porque, en general, todos los actores desean o se verán beneficiados por la implementación de estos requisitos.

3.3.3 Requisitos de información

La información que la aplicación necesita viene definida en la tabla 3.3. Aquí indicamos un identificador del requisito, el título, una descripción breve y los atributos que debe almacenar para satisfacerlo. Debido a que son términos, en general, muy relacionados con el entorno portuario, conviene definirlos a continuación:

- **Nombre**: el nombre del buque.

ID	Título	Descripción
RNF-01	Simplicidad	El sistema debe ser fácil de utilizar.
RNF-02	Acceso	El sistema debe ser accesible desde cualquier dispositivo con un navegador de internet.
RNF-03	Registro descentralizado y confiable	El sistema debe hacer uso de un registro descentralizado y confiable para guardar las escalas.
RNF-04	Contratos en lenguaje natural	El sistema debe hacer uso de tecnologías de smart contracts basadas en sintaxis textuales o lenguaje natural para facilitar su comprensión y edición.

Tabla 3.2: Requisitos no funcionales

- **MMSI**: las siglas en inglés de Identidad del Servicio Móvil Marítimo (*Maritime Mobile Service Identity*), número que identifica a cada estación de barco a efectos de seguridad y radiocomunicaciones, de acuerdo al Real Decreto 1185/2006 [15].

- **IMO**: siglas en inglés de la Organización Marítima Internacional (*International Maritime Organization*), encargada de emitir un número identificador único para cada buque, que permanece inalterable hasta que ese buque es destruido [48]. Sería el equivalente marítimo al número de bastidor de un coche.

- **Arqueo (bruto)**: de acuerdo con el Convenio Internacional sobre Arqueo de Buques, es la expresión del tamaño total de un buque [14]. El usuario podrá suministrar el arqueo directamente, en GT (*Gross Tonage*, o Tonelada de Arqueo en español), o en su defecto, la eslora, manga y puntal de trazado (todo en metros) para realizar el cálculo [41].

- **Tiempo de estancia**: tiempo que el buque pasa atracado en el puerto, desde su llegada hasta su salida.

- **Zona de atraque**: en qué parte de las definidas por el puerto solicita el atraque, ya que las tasas son distintas en función de si lo hace en el interior o en el exterior.

- **Condiciones de atraque**: el uso que se va a dar al atraque, o lo que es lo mismo, a qué supuestos de los definidos en la ley de tasas se suscribe el buque [41].

- **Número de escala**: indicador de cuantas veces el mismo buque ha atracado en el puerto, a lo largo de ese año.

- **Tipo de servicio**: un indicador de si es un buque que realiza un servicio regular (que se repite en un periodo breve de tiempo, diaria o semanalmente) o esporádico.

- **Tasa calculada**: cantidad que debe abonar el buque por su estancia en el puerto.

ID	Título	Descripción	Atributos
RI-01	Datos del buque	El sistema debe almacenar la información necesaria para identificar el buque	Nombre IMO MMSI Arqueo del buque
RI-02	Datos de la escala	El sistema debe registrar la información necesaria para el cálculo de la tasa.	Tiempo de estancia Zona de atraque Condiciones de atraque Número de escala Tipo de servicio
RI-03	Tasa	El sistema debe almacenar la cuantía a pagar.	Tasa calculada

Tabla 3.3: Requisitos de información

3.4 Conclusiones

A lo largo de este capítulo hemos comprendido cuál es la situación actual en el puerto de Santander y quienes son los actores involucrados en el registro de las escalas. Pero, más importante, hemos obtenido los requisitos que nuestra implementación debe satisfacer. No tenemos una cantidad elevada requisitos, pero son la clave para estructurar el análisis que nos dejará a las puertas de la implementación final de nuestro proyecto. Así, en el próximo capítulo, partiremos de los requisitos definidos en este para realizar el análisis y diseño de la solución.

4

Análisis y diseño de la solución

En este capítulo definiremos la solución que se dará al problema. Así, en la sección 4.1, comenzaremos indicando los casos de usos de la aplicación, a partir de los requisitos funcionales, y continuaremos modelando los datos que querremos almacenar en el sistema, en la sección 4.2, según lo indicado con los requisitos de información. Terminaremos el capítulo analizando y discutiendo algunas alternativas para la implementación del proyecto en la sección 4.3 y resumiendo las conclusiones obtenidas en la sección 4.4.

4.1 Casos de uso

La aplicación que proponemos como solución, debido también a su calidad de prueba de concepto, no es de una complejidad desmesurada. Tendremos únicamente dos casos de uso:

- El caso CU-01, registro de una escala, como vemos en la tabla 4.1. Soluciona los requisitos funcionales RF-01, RF-02, RF-04 y RF-05, según aparecen en la tabla 3.1.

- El caso CU-02, actualización del contrato, como vemos en la tabla 4.2. Soluciona el requisitos funcionales RF-03, según aparece en la tabla 3.1.

Podemos ver un diagrama de casos de uso en la figura 4.1. Como comentamos en el apartado 3.2, aunque la figura del práctico aparece definida en el modelo, no tiene ningún papel real en el sistema que estamos implementando.

Descripción	El usuario registra una petición de escala
Secuencia normal	1. El usuario entra en la aplicación web. 2. El usuario introduce todos los datos necesarios para el registro de la escala. 3. El sistema comprueba que la escala es realizable y emite el log de IALA correspondiente. 4. La escala queda registrada y el usuario es notificado de la ID del registro y de la tasa a pagar. Además, la aplicación emite el log de IALA correspondiente.
Excepciones	5. Si no hay escala disponible, el usuario es notificado. 6. Si ocurre algún otro tipo de error en el sistema, se le pide al usuario que lo intente de nuevo más tarde.

Tabla 4.1: CU-01: registro de una escala

Descripción	La autoridad portuaria actualiza las cláusulas del contrato
Secuencia normal	1. Hay un cambio en las cláusulas del contrato de las escalas. 2. La autoridad portuaria actualiza la parte textual del smart contract. 3. Se realiza un nuevo despliegue. 4. Todos los registros de escala a partir de este momento reflejan las nuevas cláusulas.
Excepciones	5. El despliegue falla. Si el error se repite la autoridad portuaria contacta con el soporte de NeptunUS.

Tabla 4.2: CU-02: actualización del contrato

4.2 Modelo de datos

Como podemos ver en la figura 4.2, el tamaño de la aplicación es bastante reducido. Se compone de:

- **Escala**: los datos específicos que necesitamos saber para una escala en concreto. Incluye:
  - `fechaLlegada` y `fechaSalida`, objetos de tipo fecha para calcular el tiempo de estancia.
  - `zonaAtraque`, la zona definida por el puerto donde se va a realizar el atraque.
  - `condicionesAtraque`, los supuestos de la ley de puertos a los que se acoje en su atraque.

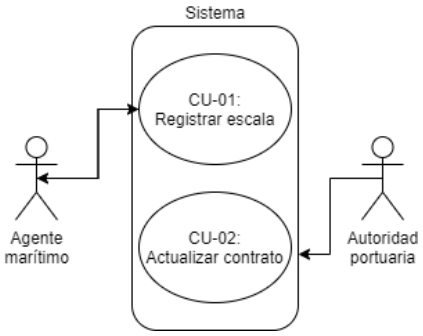

Figura 4.1: Diagrama de casos de uso

- `numeroEscala`, número de veces que el mismo buque ha atracado en el puerto durante el año.
- `tipoServicio`, si el buque realiza escalas de manera regular o esporádicamente.

- **Buque**: la información necesaria para identificar a un buque en concreto. Incluye:
  - `nombre`, el nombre del buque.
  - `MMSI`, el número único del buque según la Identidad del Servicio Móvil Marítimo.
  - `IMO`, el número único del buque de acuerdo a la Organización Marítima Internacional.
  - `arqueo`, el tamaño total del buque.
- **Registro**: la escala que queda almacenada en sí. Incluye:
  - `tasaAPagar`, cantidad a abonar por la escala. Si está vacía, indica que la solicitud de escala no ha sido autorizada.
  - `idRegistro`, el identificador único del registro de la escala.
  - `timestamp`, una string con la fecha y hora a la que se he hacho el registro.
- **Contrato**: los datos que nos permiten identificar un contrato y versión concretos. Incluye:
  - `idContrato`, el identificador único de una versión específica de un contrato concreto.
  - `nombre`, el nombre que identifica al contrato.

Para cada registro de una escala necesitaremos un buque, unos datos específicos de esa escala y el contrato por el que nos regimos. La única relación 1:1 es entre los datos y el registro, ya que un buque puede hacer varias escalas (es lo deseable de hecho) y un mismo contrato regirá muchos registros, hasta que el contrato se modifique y cambie de versión.

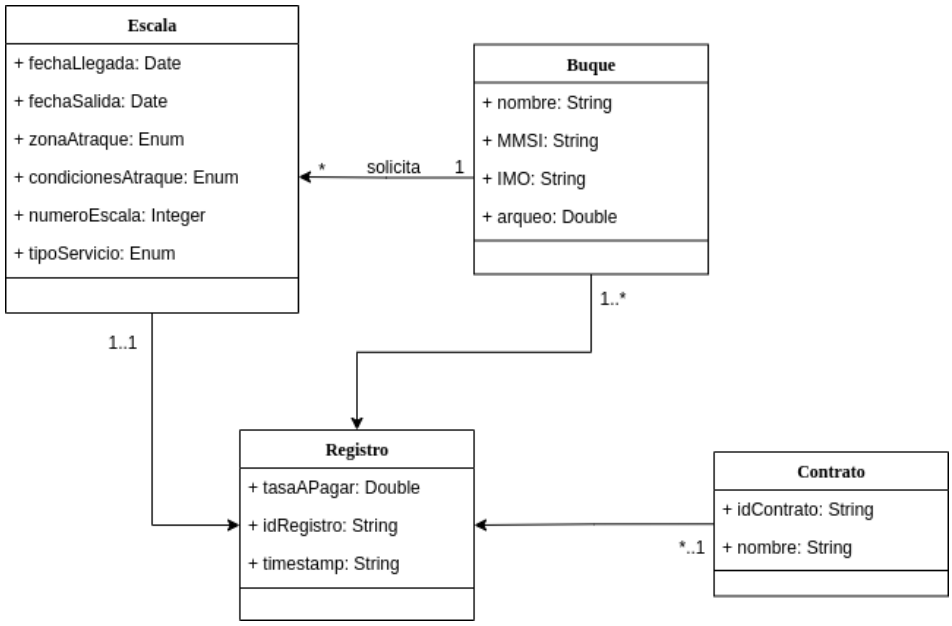

Figura 4.2: Modelado conceptual

Podemos ver el proceso de registro de una escala en la figura 4.3. Como observamos, una vez que el usuario introduce los datos del barco para el cual registra la escala, la aplicación se encarga de parsear los datos a un formato que el smart contract es capaz de entender. Después, realiza una llamada a este smart contract con dichos datos y parsea la respuesta que recibe de modo que el usuario la visualice en la aplicación. Indagaremos más en el aspecto técnico de este proceso en el capítulo 5.

4.3 Discusión de alternativas para la implementación

Aunque el capítulo 5 se dedica exclusivamente a la implementación del proyecto, consideramos que como parte del análisis que se ha hecho previo al desarrollo, está también la identificación de alternativas y la elección de cuáles se usaran. El objetivo de esta sección es por tanto describir las diferentes alternativas tecnológicas que surgieron al analizar el problema, y justificar las elecciones realizadas

Tras recoger los requisitos funcionales de la aplicación, tenemos claros tres ítems:

- La aplicación tiene que usar una blockchain.
- Para interactuar con la blockchain, necesitamos smart contracts. Y estos smart contracts tienen que ser sencillos de entender y modificar, así que debemos usar lenguajes que utilicen la sintaxis del lenguaje natural, o algo similar.
- Tenemos que poder acceder a la aplicación desde cualquier navegador de internet.

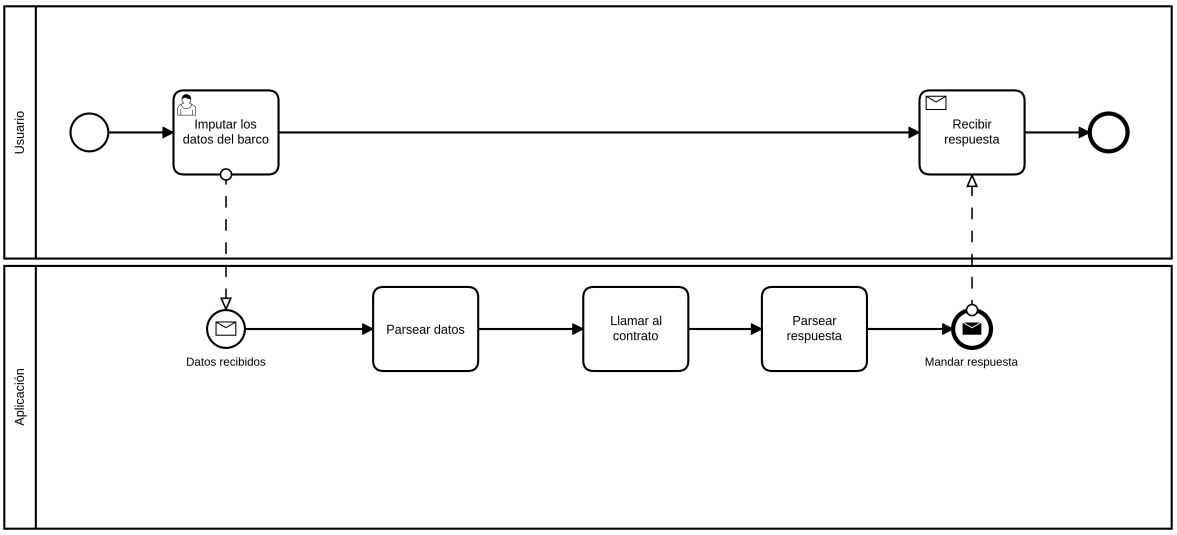

Figura 4.3: Proceso del registro de una escala

Analizando estos tres ítems, nos surgirá un cuarto a la hora de decidir cómo unir las tecnologías entre sí. Se decidirá tener un gestor de procesos como *middleware*, de modo que nuestro cuarto ítem será:

- La aplicación debe usar un gestor de procesos.

Por tanto, el resto de la sección dará respuesta a las siguientes preguntas:

1. Por qué utilizar Cicero como lenguaje para smart contracts, en la sección 4.3.1.
2. Por qué utilizar HyperLedger como blockchain, en la sección 4.3.2.
3. Por qué utilizar React.js como *frontend*, en la sección 4.3.3.
4. Por qué utilizar Camunda BPMN como gestor de procesos, en la sección 4.3.4.

4.3.1 Elección de Cicero como lenguaje de smart contracts

A pesar de que existen varias alternativas a la hora de elegir qué lenguaje de smart contracts usar en el proyecto, teníamos la importante limitación de que la sintaxis debía ser lo más parecida posible al lenguaje natural. Esto supone descartar lenguajes muy extendidos como *Solidity* [2], uno de los lenguajes más usados actualmente y exclusivo de la plataforma Ethereum pero cuya sintaxis es similar a Javascript, alejándose del lenguaje natural. También queda descartado *Bitcoin Script*, cuya sintaxis es muy compleja. Nos quedábamos en principio con dos posibles alternativas con sintaxis similar al lenguaje natural: *Cicero* [1] y *OpenLaw* [10].

La investigación nos terminó dando una respuesta rápida y casi automática. Pese a lo que el nombre pueda sugerir, *OpenLaw* resultó ser un desarrollo muy cerrado. Tan es así que, intentando obtener acceso a la documentación, se llegó a una barrera para la que se necesitaba contactar con el equipo de desarrollo para poder acceder al lenguaje. En noviembre, se mandaron varios correos electrónicos al equipo de desarrollo para obtener dicho acceso, que no recibieron respuesta. Por tanto, aunque teóricamente contábamos con alternativas, en la práctica el único lenguaje que podría resolver nuestra problemática era **Cicero**.

4.3.2 Elección de HyperLedger como blockchain

Una vez decidido el uso de Cicero como lenguaje para los smart contracts, la elección de la tecnología para la blockchain se convertía prácticamente en un mano a mano entre Ethereum [7] y la tecnología desarrollada por la Linux Foundation, HyperLedger [9]. Para la decisión final, tuvimos en cuenta dos cuestiones:

- **Facilidad de desarrollo**: prácticamente toda la documentación existente sobre smart contracts en Ethereum usaba Solidity y, del mismo modo, la mayoría de las demostraciones de Cicero usaban la otra propuesta de Linux Foundation. Por supuesto, podríamos haber investigado el uso de Cicero en Ethereum, pero HyperLedger aplanaba la curva de aprendizaje.

- **Visión empresarial**: mientras que Ethereum es una tecnología mucho más generalista, que se basa en la existencia de una criptomoneda (Ether) para mantener la red en funcionamiento, HyperLedger está más enfocada a las empresas, dando por hecho que habrá una organización detrás que mantenga el coste de la red. Esto último es más atractivo pues, en principio, pretendemos crear una blockchain que se use sola y exclusivamente en el contexto del puerto de Santander.

Teniendo en cuenta ambos puntos, finalmente, se decidió optar por **HyperLedger**.

4.3.3 Elección de React.js para la aplicación

A la hora de elegir la tecnología para desarrollar el *frontend*, es decir, la aplicación que usará el usuario real, teníamos una gran cantidad de alternativas. Al fin y al cabo, actualmente existen muchos frameworks para la creación de aplicaciones webs, todos con mucho potencial. Ejemplos de ello son React.js [11], Angular [4] o Vue.js [13].

Sin embargo, dado que este proyecto es simplemente una prueba de concepto, no se le dio mucha importancia en la investigación a qué tecnología de *frontend* utilizar. Por tanto, se eligió **React.js** por una cuestión de conocimiento: para el autor de esta memoria, Angular y Vue eran lenguajes desconocidos mientras que React era uno con el que ya había trabajado extensamente.

4.3.4 Elección de Camunda BPMN como gestor de procesos

Como ya se comentó en la introducción, la necesidad de un gestor de procesos para esta aplicación puede no entenderse intuitivamente. Y es más difícil justificarlo sin entrar en la implementación. Al fin y al cabo, no se vio la necesidad de incluir uno en el proyecto hasta que no estábamos bien introducidos en el proceso de desarrollo de la aplicación. Sin embargo, podemos entender que, en el proyecto, utilizamos tres tecnologías distintas: un framework web para que el usuario interactúe con la aplicación, una blockchain para almacenar datos y un lenguaje de smart contracts para realizar transacciones contra la blockchain.

Un gestor de procesos nos permite tener un punto intermedio entre estas tres tecnologías, orquestando las distintas actividades que se tienen que dar en un orden concreto y actuando como una suerte de intérprete entre las distintas tecnologías. Será, de hecho, el gestor de procesos el que se encargará de convertir los datos que imputa el usuario en un formato que el smart contract puede entender, el que ejecutará las transacciones del smart contract contra la blockchain y el que convertirá el resultado de la transacción en datos que se pueden mostrar al usuario.

En cuanto a la tecnología concreta, existen diferentes alternativas como por ejemplo, Camunda [6], Bonita BPM [5], o Activiti [3]. Se terminó optando por **Camunda** por la facilidad de usar, porque incluye una API que se despliega automáticamente y que podemos usar para interactuar directamente con los procesos desde una aplicación web. Además, incluye una aplicación para realizar los modelos BPMN y desplegarlos sin intermediarios, facilitando mucho el desarrollo.

4.4 Conclusiones

En este capítulo, hemos analizado los requisitos que debe satisfacer el sistema para obtener los casos de uso del mismo y definido el modelo conceptual. También hemos realizado un análisis y una posterior discusión de las alternativas que se han investigado para llevar a cabo la implementación. De este modo, en el próximo capítulo, se definirá la arquitectura de la aplicación con las tecnologías escogidas, y se detallará cómo ha sido la implementación.

5 Implementación

En este capítulo, analizaremos cómo se ha desarrollado la aplicación. Así, tras la discusión realizada en el capítulo anterior, en el apartado 4.3, comenzaremos definiendo la arquitectura del sistema. A partir de esta arquitectura, analizaremos la implementación de las diferentes partes que componen en sistema: en la sección 5.2 detallaremos la implementación del smart contract usando Cicero; en la sección 5.3, el uso de HyperLedger como plataforma de blockchain; en la sección 5.4, cómo hemos utilizado el gestor de procesos; y en la sección 5.5, la creación de la aplicación web usando React. En la sección 5.6 analizaremos el rendimiento de la aplicación, evaluando su desempeño en una serie de pruebas. Por último, concluiremos el capítulo en la sección 5.7.

5.1 Arquitectura

La arquitectura de la aplicación viene bastante marcada por el propósito que tiene como prueba de concepto. Así, necesitamos tener un *backend* donde haya una blockchain, la cual se encargue, al menos, de ejecutar el smart contract y registrar las escalas. Además, por los requisitos de acceso a la aplicación, ha de tener un *frontend* en forma de aplicación web. La conexión entre ellos se hará a través de un gestor de procesos, que funcionará como *middleware*.

Como ya adelantamos en el apartado 4.3, las tecnologías usadas son Cicero para los smart contracts, HyperLedger para la blockchain, Camunda BPMN como gestor de procesos y React.js para el *frontend*. Podemos ver un esquema informal de la arquitectura de la solución en la figura 5.1, y una definición más formal de la arquitectura en la figura 5.2. Podemos comprobar cómo el gestor de procesos actúa como centro de la aplicación, proveyendo de interfaces a la distintas partes de la aplicación para que se comuniquen entre ellas. Además,

existe en este diagrama una relación de dependencia entre blockchain y smart contract, pues los smart contracts son necesarios para realizar transacciones contra la blockchain.

En las siguientes secciones hablaremos más en profundidad de las distintas tecnologías, y analizaremos la implementación.

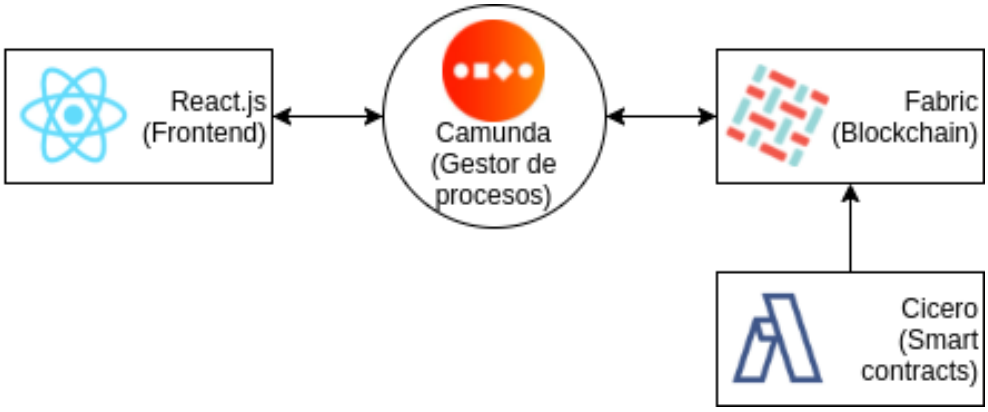

Figura 5.1: Esquema de la arquitectura de la solución

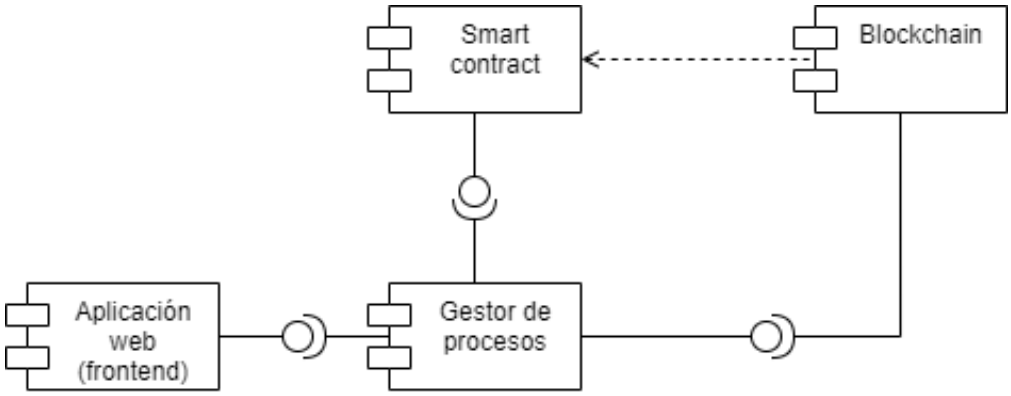

Figura 5.2: Diagrama de componentes de la solución

5.2 Smart contracts: Cicero

El smart contract se encargará exclusivamente del cálculo de la tasa, ya que así tendremos los datos que necesitamos y realizaremos la transacción que queremos registrar en la blockchain. En la plataforma Cicero, los contratos se despliegan en un formato propio llamado Cicero Template Archive (.cta). Estos son archivos comprimidos que contienen un directorio en el cual deben aparecer los ficheros que se detallan en la tabla 5.1. En dicha tabla indicamos también en qué sub-apartado de esta sección hablaremos con más detalle de cada uno de esos archivos. Así, en la sección 5.2.1 hablaremos del texto y gramática del

smart contract; en la sección 5.2.2, del modelo; en la sección 5.2.3, de la lógica; y en la sección 5.2.4, de la estructura de una petición al smart contract.

Archivo	Descripción	Sub-apartado
`sample.md`	Contrato completo	5.2.1
`grammar.tem.md`	Gramática del contrato	5.2.1
`model.cto`	Modelo del contrato	5.2.2
`logic.ergo`	Lógica del contrato	5.2.3
`request.json`	Petición al contrato	5.2.4

Tabla 5.1: Ficheros necesarios en un smart contract de Cicero

5.2.1 Texto y gramática del contrato

Lo primero que necesita el contrato es el texto del contrato en sí. Necesitaremos un archivo en formato *Markdown*, con una sintaxis particular conocida como *TemplateMark*. En este archivo, llamado `grammar.tem.md`, es donde definimos la gramática del contrato, o lo que es lo mismo, qué caracteres del contrato equivalen a qué variable del contrato. Vemos un ejemplo de este archivo en el listado 5.1, donde podemos leer la sintaxis de un artículo completo.

Listado 5.1: Un artículo del contrato con el formato de la gramática

```
1 ## Articulo 202. Cuantías básicas.
2     El valor de las cuantías básicas de la tasa del buque (B y S) se
        establece para todas las Autoridades
3 Portuarias en {{cuantiaB}}€ y {{cuantiaS}}€, respectivamente. Estos
    valores podrán ser revisados en la Ley de
4 Presupuesto Generales del Estado o en otra que, en su caso, se apruebe
    a estos efectos en función de la
5 evolución de los costes portuarios, logísticos y del transporte, así
    como de los productos transportados,
6 tomando en consideración las necesidades asociadas a la competitividad
    del nodo portuario y de la economía.
```

Del mismo modo, en el archivo `sample.md` es donde escribimos el contrato tal cual, también en formato *Markdown*, pero con todos los datos vigentes. Podemos ver un ejemplo en el listado 5.2. Si comparamos ambos archivos vemos que son idénticos en todo, carácter por carácter, salvo en los lugares donde se declaran las variables: en el primer archivo con dos llaves (`{{}}`) y dentro el nombre de la variable, en el segundo leemos directamente el valor. De este modo, dado el caso de que queramos cambiar un valor (en este ejemplo podrían ser las cuantías B o S) simplemente tendremos que cambiarlo en el archivo `sample.md`.

Listado 5.2: El mismo artículo del listado 5.1 pero con la información del contrato

```
1 ## Artículo 202. Cuantías básicas.
```

```
2     El valor de las cuantías básicas de la tasa del buque (B y S) se
          establece para todas las Autoridades
3 Portuarias en 1.43€ y 1.20€, respectivamente. Estos valores podrán ser
      revisados en la Ley de
4 Presupuesto Generales del Estado o en otra que, en su caso, se apruebe
      a estos efectos en función de la
5 evolución de los costes portuarios, logísticos y del transporte, así
      como de los productos transportados,
6 tomando en consideración las necesidades asociadas a la competitividad
      del nodo portuario y de la
7 economía.
```

5.2.2 Modelo

En el archivo `model.cto` recogemos el modelo tanto del contrato como de las peticiones que esperamos y las respuestas que mandaremos. Estos modelos son definiciones de las variables esperadas, en las que especificamos tanto su tipo (`Boolean`, `String`, `Integer`, `Double`, `Long` o `DateTime` [44]) como su nombre.

Dentro del modelo podemos tener dos tipos de declaraciones: *assets*, que definen los tipos de variables que ya vienen de un contrato, y *transactions*, donde declaramos las variables que aparecen en los JSON de petición y respuesta. Para verlo de otro modo, si estuviéramos trabajando con una API común, es en el modelo donde definimos qué datos vamos a coger de nuestra base de datos (los datos del contrato en nuestro caso), qué nos llega en la petición y qué datos vamos a devolver en la respuesta. Para nuestra implementación definiremos, además de todas las variables del contrato en el *asset* `ContratoTasas`, tres *transactions*:

- `requestZonaI`: donde definimos todo lo que necesitaremos para una petición de atraque en la Zona I, o en los casos especiales tanto de la Zona I como de la Zona II
- `requestZonaII`: donde definimos lo necesario para una petición de atraque en la Zona II.
- `respuestaTasas`: donde definimos qué atributos aparecerán en el JSON que mandaremos como respuesta.

En el listado 5.3 vemos un ejemplo de las declaraciones de variables. En este caso vemos sólo las declaraciones de `requestZonaII` y `respuestaTasas`, junto al comienzo del modelo de `ContratoTasas`. Según la clase que extienden, se indica si son las variables que esperamos en la petición (`Request`), las que mandaremos en la respuesta (`Response`) o las que obtendremos del propio contrato (`AccordContract`).

Listado 5.3: Ejemplo de declaración de variables en el modelo

```
1 transaction requestZonaII extends Request {
2   o Boolean usarB
3   o Double arqueo
```

```
4   o Double horas
5   o Integer escala
6   o Boolean servicioRegular
7   o Boolean A1II
8   o Boolean A2II
9   o Boolean A3II
10  o Boolean B1II
11  o Boolean B2II
12  o Boolean B3II
13 }
14
15 transaction respuestaTasas extends Response {
16   o Double tasaAPagar
17 }
18
19 /**
20  * The model for the contract
21  */
22 asset ContratoTasas extends AccordContract {
23   o Double arqueoMinimo
24   o Double horasMinimas
25   o Double horasMaximas
26   o Double cuantiaB
27   o Double cuantiaS
28   o Double coefCorrector
29   o Double coefA1
30   o Double coefA2
31 ...
```

5.2.3 Lógica

En el archivo `logic.ergo` es donde definimos el comportamiento del contrato como tal. Para ello, hemos de usar el lenguaje **Ergo** cuya documentación podemos encontrar en [42]. Es semánticamente similar a JavaScript aunque muy simplificado, hecho que se puede deber a que está aún en desarrollo.

Se basa en la declaración de cláusulas, para las que necesitamos indicar qué *request* de las declaradas en el modelo la llama y qué response se devuelve. Cada *request* se puede usar únicamente con una cláusula, pero el mismo tipo de *response* se puede devolver en distintas cláusulas. Estas son las funciones que utiliza el lenguaje, por lo que no podemos declarar ninguna función si no es llamada por una cláusula. Del mismo modo, desde una cláusula no podemos llamar a otra, toda la lógica debe estar contenida en la misma cláusula. Es por ello que optamos por la solución de utilizar dos *requests* distintas para cada zona, ya que de esa manera se discrimina qué funcionalidad es necesaria antes de hacer la llamada.

Las variables son siempre constantes aunque se pueden declarar tanto como `const` como `let`. No permite que se declare el tipo de la variable aunque es un lenguaje fuertemente tipado, de modo que operar con variables de tipos distintos nos dará error. A nivel de lógica,

permite comparaciones simples (mayor, menor o igual que) y decisiones de tipo if/else, aunque no permite hacer bucles. Incluye también la creación y el manejo de excepciones aunque a la fecha de realizar la implementación era una característica recién añadida que no funcionaba correctamente, así que no la pudimos usar en nuestro proyecto.

Acercándonos ya a nuestro caso particular, dado que es una lógica relativamente simple, podemos resumirla en el pseudocódigo que aparece en el listado 5.4. Simplemente queremos multiplicar la tasa base por todos los coeficientes que aplican a una escala en particular.

Listado 5.4: Lógica del cálculo de las tasas (en pseudocódigo)

```
1 tasa = arqueo * horas * coeficienteCorrector
2
3 for each (clausulaEnPeticion) {
4     tasa = tasa * coeficientesClausulas[clausulaEnPeticion]
5 }
6
7 return tasa
```

En la realidad, debido a las limitaciones de Ergo, es un poco más complejo. Vemos un ejemplo recortado en el listado 5.5. Lo que hacemos es multiplicar la parte básica del cálculo de la tasa (arqueo por horas) por todas aquellas tasas que aparezcan como `true` en el JSON de la petición. Ergo no nos permite mucha más flexibilidad. No obstante, una leve lógica, por ejemplo, para tener en cuenta el arqueo mínimo distinto en cualquiera de los supuestos de la letra E en la Zona I (lo vemos en la condición que comienza en la línea 7 en el listado 5.5), o para que devuelva directamente la tasa en el caso de la letra F de la misma zona.

Listado 5.5: Fragmento de la lógica del contrato

```
1  clause tasaZonaI(request : requestZonaI) : respuestaTasas {
2
3      let horas = request.horas;
4
5      let arqueo = if request.arqueo > contract.arqueoMinimo
6      then request.arqueo
7      else if (request.E1 or request.E2 or request.E3 or request.E4 or
           request.E5 or request.E6 or request.E7 or request.E8 or request
           .E9) and request.arqueo < contract.arqueoMinimoE
8      then contract.arqueoMinimoE
9      else contract.arqueoMinimo;
10
11     let preTasa0 = horas * arqueo * contract.coefCorrector;
12
13     let preTasa1 = if request.usarB and !(request.H1 or request.H2 or
           request.I)
14     then preTasa0 * contract.cuantiaB
15     else preTasa0 * contract.cuantiaS;
16
17     ...
18     ...
```

```
19
20      let tasaAPagar = if request.descuentoZonaII
21      then tasaSinDescuento * contract.descuentoZonaII *
            descuentoEscalaFinal
22      else if request.descuentoZonaI
23      then tasaSinDescuento * contract.descuentoZonaII *
            descuentoEscalaFinal
24      else tasaSinDescuento * descuentoEscalaFinal;
25
26      return respuestaTasas{tasaAPagar: tasaAPagar}
27  }
```

5.2.4 Petición

Algo que no es necesario dentro del contrato pero que sí lo será para interactuar con el mismo es el archivo de *request*. Esta es la parte menos reseñable de Cicero, pues realmente se comporta como una petición a una API.

Así, tenemos un JSON como el del listado 5.6. Lo primero que se detalla, con la variable `$class`, es la request a la que estamos contactando, que en nuestro caso se corresponde con la zona en la que queremos atracar. A continuación, aparecen todos los datos que especificamos en el modelo para dicha request. Hay que tener en cuenta que son obligatorios, y que en caso de que falte alguno o especifiquemos un valor de tipo distinto al declarado, la ejecución dará error y no obtendremos respuesta.

En nuestro caso, y debido a que los supuestos no son comprobables de alguna manera cuantificable (lecturas que podamos obtener de un sensor, por ejemplo), en cada una de esta peticiones esperamos un número de variables de tipo `boolean`, que nos indica qué letras de la ley de tasas está cumpliendo el buque en su atraque. Las compatibilidades entre ellas son controladas en el formulario desde el que se lanza la petición.

Listado 5.6: Ejemplo de petición al contrato

```
1  {
2      "$class": "org.santanderPort.requestZonaII",
3      "usarB": true,
4      "arqueo": 150.0,
5      "horas": 15.0,
6      "escala": 12,
7      "servicioRegular": false,
8      "A1II": true,
9      "A2II": false,
10     "B1II": false,
11     "B2II": false,
12     "B3II": false
13 }
```

5.3 Blockchain: HyperLedger Fabric

Como hemos introducido en la sección 5.1, HyperLedger [9] es una tecnología basada en blockchain nacida en 2015 bajo el brazo de la Linux Foundation, junto a otras empresas de finanzas, IoT o banca como la cadena de supermercados estadounidenses Walmart, Intel o IBM, que es una de las principales colaboradoras [39]. Hyperledger tiene varias líneas de trabajo paralelas. Una de ellas, Burrow, se basa en Ethereum para la ejecución de smart contracts con Solidity. En nuestro caso, usamos una de las más conocidas, **Fabric**. HyperLedger Fabric es una tecnología modular de blockchain desarrollada en Go, y especialmente conocida y utilizada por su flexibilidad [30]. Se compone de:

- El **ledger**, que es la blockchain en si, la cadena de bloques en la que se almacenan las transacciones así como los datos que incluyen dichas transacciones. Es un conjunto de datos en el que se puede confiar, ya que todos los datos se comprueban antes de añadirse y se vuelven inmutables tras la comprobación. Está formada por pares de clave y valor, de modo que se pueden consultar todos los valores de una determinada clave.

- La **state database**, una base de datos que contiene el último valor de cada una de las claves. Es una forma de acceder rápidamente al valor más actual de una clave, algo que nos interesa si queremos saber, por ejemplo, cuántas escalas ha realizado hasta el momento un buque.

- Los **chaincodes**, los smart contracts propios de HyperLedger, que realmente actúan como scripts que permiten desplegar aplicaciones en la blockchain.

La flexibilidad de la que hablábamos anteriormente se debe a que HyperLedger permite que tanto la state database como los chaincodes utilicen tecnologías y lenguajes diversos, no impone ninguno. Además, tiene dos características fundamentales:

- Es una plataforma *permisionada*, de modo que no puede acceder cualquiera, sino que es necesario tener autorización para ello. Por el contrario, en una plataforma pública, cualquier persona tendría acceso tanto al registro de las transacciones como a la posibilidad de participar en ella, facilitando la suplantación de identidad. El que se necesite autorización para acceder a esta blockchain hace que la propia plataforma actúe como capa de confianza.

- No hace uso de una criptomoneda nativamente (aunque se puede implementar una), de modo que las transacciones no tienen un valor per se. Esto reduce el riesgo de ataque a la red, ya que no se obtiene un beneficio económico al hacerlo.

Estas características se deben a que es una plataforma muy orientada a empresas, más que a usuarios finales, ya que se centra más en aportar las ventajas de blockchain a las

comunicaciones entre clientes, y no tanto a permitir que particulares puedan usarla para aplicaciones individuales [39].

En cuanto a la implementación, tenemos la suerte de que HyperLedger en general y Fabric en particular es un proyecto open source muy bien mantenido. Así, en la propia documentación de HyperLedger Fabric encontramos un repositorio en GitHub [36] donde podemos acceder a todos los scripts que nos permiten iniciar una red de prueba local de la blockchain, además de para poder crear nuestra propia red en producción. De igual modo, Accord Project nos facilita otro repositorio [43] donde se encuentra el Chaincode que nos permitirá desplegar smart contracts en el formato de Cicero dentro de esta red.

Para el despliegue de la blockchain se creó un script de despliegue que permite levantar una red de prueba activa y lista para ejecutar nuestras peticiones. Para dicho despliegue se sigue el proceso que aparece en la figura 5.3, que recoge los siguientes pasos:

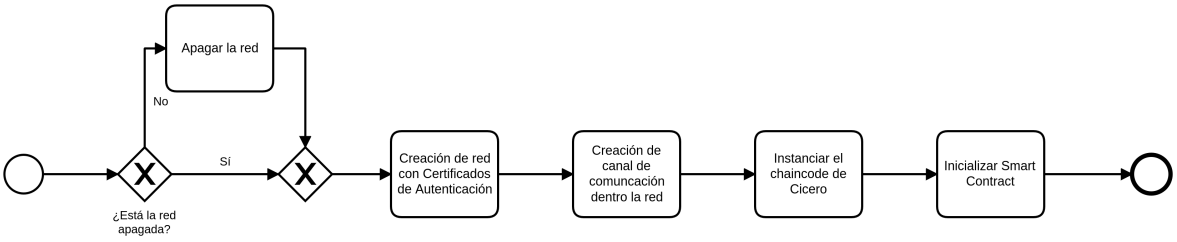

Figura 5.3: Proceso de iniciar la red local de prueba

1. Intentamos apagar la red. Si no hay ninguna activa no pasará nada, pero si hay alguna ya activa, nos permitirá evitar conflictos.

2. Creamos una red con certificados de autenticación, lo que nos permite que nuestra red actúe como capa de confianza.

3. Creamos un canal de comunicación dentro de la red. Esto es necesario para que más adelante podamos instalar el Chaincode de Cicero. Aquí ya tenemos una red de prueba de HyperLedger Fabric activa. Sin embargo, como no hemos instanciado ningún Chaincode, no podemos hacer nada con ella todavía.

4. Instanciamos el Chaincode de Cicero. Debido a que este Chaincode en concreto está desarrollado en Node.js, lo instalaremos usando npm antes de instanciarlo, para poder llamar al *script*.

5. Inicializamos en esta instancia un smart contract de Cicero.

Una vez hemos terminado todos estos pasos, ya tenemos una red de prueba en la que hay instanciado un smart contract al que podemos realizar peticiones. Estas peticiones se harán también a través de uno de los scripts que nos facilitaba el repositorio de Accord Project. Se planteó implementar también una función de búsqueda en la blockchain. Sin embargo,

debido al funcionamiento de la red de prueba, nos resultó prácticamente imposible el poder acceder a los nodos, y aún más realizar consultas. Queda claro cómo hacerlo con nodos reales en máquinas distintas pero, en su versión virtualizada, no se pudo llevar a cabo la implementación de esta característica.

5.4 Gestor de procesos: Camunda BPMN

Camunda es una plataforma de gestión de procesos basada en diagramas BPMN (Business Process Model and Notation, notación y modelado de procesos de negocio en español) [31]. Es un stack compuesto de diversas aplicaciones que incluyen:

- **BPMN Workflow Engine**: el motor que ejecuta los procesos en si, encargado de entender el modelo BPMN y realizar las tareas en el orden y tiempo que indica, así como de gestionar las esperas o los datos introducidos en formularios. Para cada proceso desplegado, existe una API que nos permite interactuar con él: inicializar una instancia, detenerla, consultarla...
- **Modeler**: la aplicación que permite crear los diagramas BPMN, además de convertirlos en procesos que el Workflow Engine puede ejecutar.
- **Cockpit**: una aplicación web que permite monitorizar los flujos de trabajo e inspeccionar instancias mientras están en ejecución.

Estas tres aplicaciones son las que hemos utilizado para nuestro caso concreto, aunque el paquete incluye más. Por ejemplo, **Tasklist**, que permite que los usuarios finales ejecuten las tareas directamente, o **DMN Engine**, que automatiza la toma de decisiones.

La suite de Camunda tiene unas miras muy empresariales, preparada para ejecutar procesos de negocios de gran tamaño, automatizando todo lo posible y buscando que las tareas manuales sean lo más eficientes posibles. Sin embargo, nuestro modelo (figura 5.4) es muy sencillo. En este modelo existe un único actor y sólo hay una condición que lleva a la ejecución o no de una tarea.

Tener un gestor de procesos actuando como *middleware* nos permite ganar modularidad y orquestar la lógica del sistema de manera sistemática. Al no implementar lógica (más allá de la validación) en el *frontend*, podemos sustituirlo o incluso cambiar la plataforma. Lo único que necesitaremos para iniciar el proceso será hacer una llamada a la API de Camunda con los datos que el usuario haya introducido.

Nuestro uso del potencial que tiene Camunda como gestor de procesos se centra en crear tres scripts en JavaScript, llamados *workers* en el propio vocabulario de Camunda. Cada uno de ellos ejecuta un método que escucha las llamadas que hace el motor a las diversas tareas del proceso. Atendiendo de nuevo a la figura 5.4, la ejecución de nuestro proceso tendrá la siguiente traza:

1. El motor recibe una llamada que inicializa una instancia del proceso con los datos que el usuario ha introducido en el *frontend*.

2. Se comprueba la disponibilidad para el tipo de escala y fecha solicitada. Tenemos una bandera que pondremos a `true` si efectivamente hay escala disponible, `false` en caso contrario. En esta prueba de concepto, al no tener una base de datos como tal, decidimos aleatoriamente si hay escala disponible o no, con 80 % de probabilidad de que haya (este *worker* sería parsear-datos.js). Además, tenemos en el propio código una lista de buques del que cogemos uno al azar para simular el login del usuario. En la aplicación real, obtendríamos la información del buque en función de qué usuario está autentificado.

3. En el caso de que haya escala, continuaremos hacia la tarea que se encarga de llamar a la blockchain para registrar la escala y calcular la tasa a pagar. En caso contrario, seguimos adelante sin hacer nada más (este *worker* sería registro-escala.js).

4. Por último, devolvemos al frontend una notificación con el resultado de la operación: si había escala, cuánto hay que pagar, y si no había, que lo intente con otras fechas. Si hubiera un error en la ejecución, también se le notificaría al usuario. En la prueba de concepto, esta es la parte menos modular, ya que lo hicimos muy concreto para escribir en una pseudo-base de datos donde el frontend espera las notificaciones. Sin embargo, de cara a un entorno de producción, también se podría modularizar haciendo que, por ejemplo, devuelva una llamada a una API cuya existencia requiramos al frontend (este *worker* sería notificar-usuario.js)

Cada uno de estos *workers* genera una serie de logs en un sistema similar al de IALA (International Association of Lighthouse Authorities), la asociación que se encarga de regular las comunicaciones marítimas.

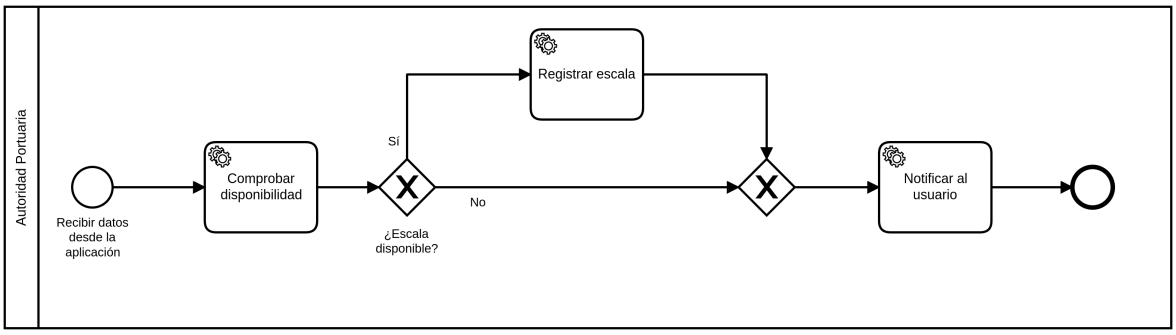

Figura 5.4: Diagrama del proceso de registro de una escala

5.5 Frontend: React.js

React no es estrictamente un *framework* sino una biblioteca que permite construir interfaces de usuario [26]. Se suele usar junto a ReactDOM (un paquete de React que, entre

otras cosas, permite renderizar componentes) como frameworks de desarrollo web a base de piezas de código lógicas y autocontenidas, los llamados componentes. Es muy similar a JavaScript, aunque expande su sintaxis para permitir también la utilización de código parecido a HTML.

El *frontend* es, de todo lo implementado en la prueba de concepto, lo técnicamente menos interesante. Se eligió React como se podría haber elegido Angular, Vue, PHP o cualquier otra tecnología que nos permita crear un formulario, validarlo, llamar a una API con los datos de dicho formulario y recibir una respuesta. No existe en este contexto una ventaja clara de uno frente al resto, fue simplemente por comodidad al tener más conocimiento de React que del resto. Dado que simplemente iba a ser necesario un formulario para capturar los datos, no se llegaron a realizar mockups, sino que directamente se pasó a implementar dicho formulario para recoger los datos especificados en el apartado 4.2.

Lo único destacable de la implementación es el hecho de que tenemos un archivo de texto en formato JSON, que utilizamos para la comunicación desde Camunda hacia el usuario. Esto se hizo así porque, realmente, el frontend no deja de ser una comodidad para comprobar el funcionamiento de nuestra solución, pero no deja de ser un prototipo. Por ejemplo, sabemos que no va a haber problemas de concurrencia porque sólo hay un usuario a la vez, pero de cara a una aplicación real, React nos permitiría implementar una API a la que un *worker* de Camunda pueda llamar para realizar una notificación. Sin embargo, este punto quedaba fuera de lo que queremos demostrar.

Detallando la implementación que hemos hecho, para mandar el resultado tenemos un formulario con validaciones simples, como que varios campos deben de no estar vacíos para poder enviarlo. La única validación algo más compleja que encontramos es la de la fecha y hora de salida y llegada (la fecha de salida no debe ser anterior a la de llegada), que encontramos en el listado 5.7. Lo que hacemos es, cuando se actualiza o bien la fecha o la hora, ya sea de salida o de llegada, forzamos a que se actualice el estado usando el hook `useState`. Después, llamamos a uno de estos métodos (en el listado vemos el ejemplo concreto de la fecha de salida, pero hay uno para cada uno de los inputs), donde convertimos la información que tenemos a objetos de tipo fecha y las comparamos. Si la comparación es correcta, actualizamos el nuevo campo en el estado. En caso contrario, ponemos a `true` una bandera que indica que hay un error en la fecha, haciendo que se muestre el mensaje en rojo en la pantalla.

Listado 5.7: Ejemplo de validación de las fechas en los formularios

```
const setAndCheckFechaSalida = (fecha) => {
    var dateLlegada = new Date(`${fechaLlegada}T${horaLlegada}:00Z`);
    var dateSalida = new Date(`${fecha}T${horaSalida}:00Z`);
    if(dateLlegada.getTime() >= dateSalida.getTime() ){
        setErrorFecha(true);
        setFechaSalida('');
    } else {
        setErrorFecha(false);
        setFechaSalida(fecha);
    }
```

```
11 }
```

En el momento que mandamos el formulario, hacemos una petición a la API de Camunda, como vemos en el listado 5.8, usando para ello un método asíncrono (`fetch()`). Tras esto, iniciamos una espera. Dado que sabemos la ID de la llamada que hemos hecho a la API de Camunda, esperamos que el último de los *workers* definidos en la sección 5.4 actualice el JSON donde vamos almacenando los resultados. Esto no deja de ser una implementación muy sencilla de una base de datos, en la que usamos como clave primaria la ID de la llamada a la API. En el momento en que dicha clave aparece en este JSON, cargamos los resultado que ahí aparecen. Si ocurre algún error, y el JSON no se ha actualizado en un máximo de 9 segundos, detenemos la ejecución y le indicamos al usuario que ha ocurrido un error. Este tiempo se eligió ya que, después de un proceso de prueba y error, se determinó que era el *timeout* mínimo que se necesitaba para asegurar que la blockchain tenía tiempo para responder sin que la espera se demorara excesivamente. De cara a escalar la aplicación, este tiempo de *timeout* debería revisarse en función a las característica del hardware donde la blockchain se aloje.

Listado 5.8: Petición a la API usando un método asíncrono

```
1  useEffect(() => {
2      if(request.loading){
3          const requestOptions = {
4              method: 'POST',
5              headers: { 'Content-Type': 'application/json' },
6              body: JSON.stringify(obj)
7          };
8          fetch('engine-rest/process-definition/key/Process_0del55l/start
               ', requestOptions)
9              .then(response => response.json())
10             .then(data => {
11                 console.log(data.id);
12                 setRequest({loading: false, id: data.id});
13             });
14         }
15     }, [request, setRequest, obj]);
```

En las figuras 5.5, 5.6 y 5.7 podemos ver un mismo ejemplo, la primera página de la aplicación, desde un PC, tablet y móvil respectivamente. Ya que React implementa un diseño responsivo de forma nativa, conseguimos tener una aplicación compatible con prácticamente cualquier dispositivo con navegador web de forma directa. En el Anexo B podemos ver más imágenes de los resultados y los errores, así como una visión más detallada del funcionamiento del *frontend* desde el punto de vista del usuario.

Figura 5.5: Ejemplo de la web vista desde un PC

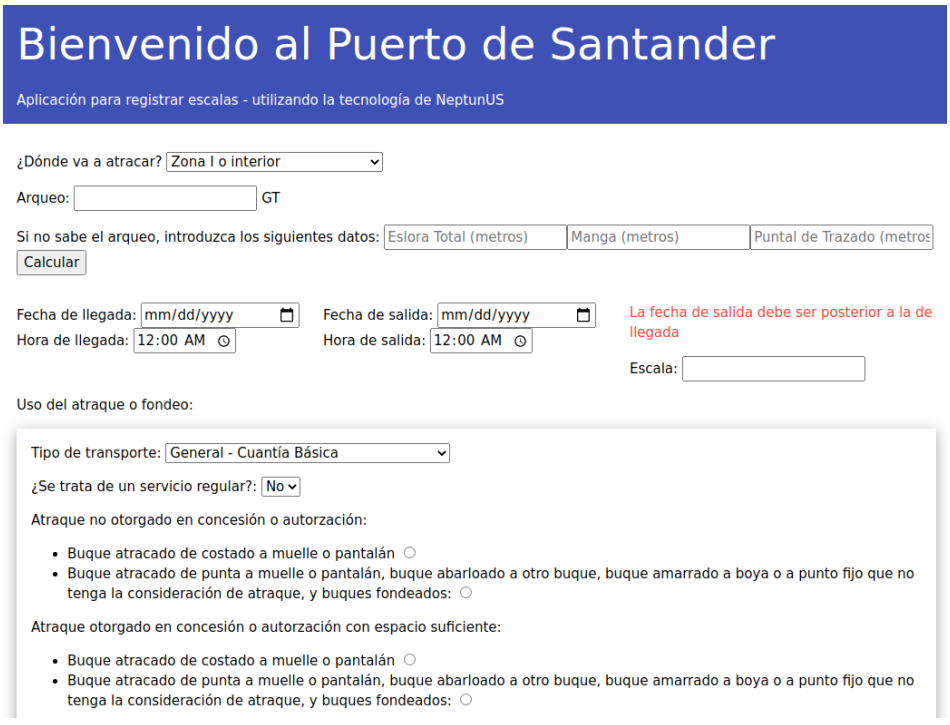

Figura 5.6: Ejemplo 5.5 desde una tablet

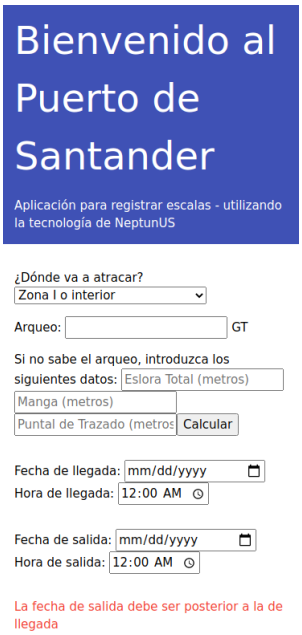

Figura 5.7: Ejemplo 5.5 desde un móvil

5.6 Evaluación y pruebas

Es indudable que el uso de una blockchain conlleva un aumento en el tiempo que necesitamos para guardar un dato. Frente a una base de datos normal, relacional o no, que en el peor caso puede tener que validar algunos datos antes de guardarlos, una blockchain necesita realizar toda una serie de cálculos matemáticos para garantizar la trazabilidad de los datos.

Para comprobarlo, realizamos doce ejecuciones diferentes en nuestra aplicación. El ordenador en el cual se hicieron las pruebas fue siempre el mismo, con un procesador Intel Core i7-8550U a 1.8GHz y 16GB de memoria RAM. El formulario se cumplimentó con datos aleatorios, intentando distribuirlos igualitariamente entre los cuatro tipos de petición que podemos realizar.

Podemos ver los datos de las doce ejecuciones en la tabla 5.2, incluyendo el número de ejecución, el tiempo que tardó (en segundos), el resultado obtenido y para que zona se intentó hacer la escala. A partir de esta tabla obtuvimos los valores medios, recogidos en la tabla 5.3, donde podemos ver el tiempo medio en segundos para cada zona, para cada resultado, y en total. La figura 5.8 contiene una representación gráfica de esta última tabla, representando para cada zona el tiempo medio en segundos si hay escala (en azul), si no la hay (en rojo) y el tiempo medio (en naranja).

La media general es de alrededor de seis segundos pero tenemos que tener en cuenta que, cuando no hay escala, nos saltamos toda la tarea que realiza en sí el registro en la blockchain. Descartando esos casos, vemos que el tiempo medio es de 8.93 segundos, y que

todos los valores están en torno al 8.90, sin que ninguno destaque por encima o por debajo. El peor caso es el de la ejecución 11, y el mejor el de la ejecución 6.

Cabe recordar que esto es un entorno de pruebas en el que las comunicaciones entre nodos se hacen en un contexto virtualizado, y que al ser un prototipo, sólo hay una persona a la vez realizando consultas. En el momento en el que un nodo de la blockchain tenga que dar respuesta a varias peticiones a la vez, tenemos que esperarnos incluso peores resultados. Al fin y al cabo, una blockchain no está preparada para ser una base de datos que soporte concurrencia, sino un libro de contabilidad, que puede permitirse el hacer las operaciones de forma lineal, conforme le llegan.

Ejecución	**Tiempo (segundos)**	**Resultado**	**Zona**
1	8.96	Hay escala	Zona I
2	1.12	No hay escala	Zona I dique exento
3	8.90	Hay escala	Zona II atraque
4	8.92	Hay escala	Zona I
5	0.97	No hay escala	Zona II fondeo
6	8.87	Hay escala	Zona II atraque
7	8.89	Hay escala	Zona I dique exento
8	8.94	Hay escala	Zona II fondeo
9	1.07	No hay escala	Zona II fondeo
10	1.02	No hay escala	Zona I
11	8.97	Hay escala	Zona II atraque
12	8.89	Hay escala	Zona I dique exento

Tabla 5.2: Ejecuciones consecutivas y tiempo para obtener respuesta

Zona	**Tiempo medio si hay escala (segundos)**	**Tiempo medio si no hay escala (segundos)**	**Tiempo medio (segundos)**
Zona I	8.94	1.02	6.30
Zona I dique exento	8.89	1.12	6.30
Zona II atraque	8.94	1.07	6.31
Zona II fondeo	8.96	0.97	6.30
Total	8.93	1.05	6.30

Tabla 5.3: Medias de tiempo de ejecución

5.7 Conclusiones

Este capítulo es el más técnico de todos los que incluye la memoria. En él, hemos definido la arquitectura que ya habíamos decidido al diseñar la aplicación, y después hemos analizado la implementación de cada una de las partes, indicando lo más interesante del código de

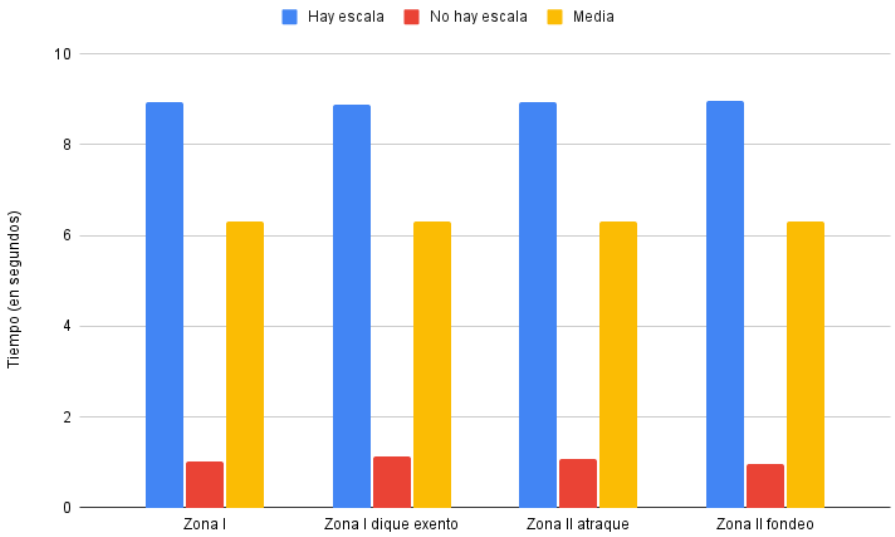

Figura 5.8: Visualización gráfica de la tabla 5.3

cada una de las tecnologías utilizadas. Para terminar, hemos realizado una evaluación no exhaustiva de la implementación, llegando a la conclusión de que los resultados no son muy esperanzadores de cara a una posible aplicación real de este proyecto.

6 Conclusiones

En este capítulo, se recapitulará el trabajo realizado y se analizarán los resultados obtenidos y si se ha alcanzado o no el objetivo. En la sección 6.1 dibujaremos unas conclusiones generales sobre el proyecto y, a continuación, en la sección 6.2, indicaremos cuál debería de ser el sendero a seguir de cara a futuros proyectos que partan de éste. Para terminar, en la sección 6.3 se hace una reflexión sobre conceptos que se han ido desarrollando en el mundo a lo largo de la realización de este proyecto, y que le afectan de manera directa.

6.1 Conclusiones generales

Remontándonos a la sección 1.3, definimos varios objetivos para el proyecto. El primero, que era introducirnos en el uso de blockchain y smart contracts, lo podemos dar claramente como realizado. El segundo y el tercero, que buscaban implementar una infraestructura que use blockchain y smart contracts para registrar datos y usarla para la digitalización del proceso de registros de escalas en el puerto de Santander, también se han alcanzado, pero debemos matizar los resultados.

Efectivamente, hemos conseguido realizar una implementación que nos ha servido como prueba de concepto. Un usuario final no debería usar la solución que hemos terminado realizando en el proyecto, pero ha quedado demostrado que efectivamente blockchain es una tecnología que se puede usar para el registro de los datos específicos de las escalas de buques. Hemos terminado teniendo una aplicación web con un almacén de datos seguro, descentralizado y con trazabilidad. Sin embargo, hay un factor que no tuvimos en cuenta a la hora de definir los objetivos: el rendimiento, analizado en el apartado 5.6. Tenemos una implementación que funciona, pero en ese apartado hemos concluido que el rendimiento es bastante peor que el que tendría una aplicación al uso y que al escalar la aplicación,

podemos esperar que empeore. Al fin y al cabo, es indudable que el uso de una blockchain conlleva un aumento en el tiempo que necesitamos para guardar un dato. Habría por tanto que analizar si las ventajas que obtenemos compensan la pérdida de rendimiento.

6.2 Trabajo futuro

Ya que este proyecto no ha terminado con la realización de una aplicación final, consideramos necesario apuntar la dirección de futuros proyectos que puedan surgir a partir de este. Sobre todo porque consideramos que, antes de escalar este proyecto hacia una aplicación real, hay ciertas dudas que deberían ser despejadas:

1. Si es posible o no añadir algunas características de blockchain a otro tipo de bases de datos, en especial la trazabilidad. Sería interesante realizar una prueba de concepto similar a esta en la que se intente resolver el mismo problema sin una blockchain, y comparar los resultados con los obtenidos aquí.

2. El funcionamiento en sí del sistema que incluya una blockchain. En general, es una tecnología que se basa en tener una gran cantidad de nodos para conseguir la descentralización, no es algo que podamos tener en pocos servidores. Por tanto, habría que definir el alcance. ¿Será un proyecto de una comunidad autónoma, nacional o internacional? ¿Cuántos nodos necesitaremos? ¿Serán privados (sólo las autoridades portuarias son nodos) o públicos (cualquier persona puede añadir un equipo que funcione como nodo)? ¿Querremos crear nosotros nuestros propios nodos, o nos bastarán con servicios en la nube como Azure o Amazon Web Services? En el caso de que sean públicos, tendremos que decidir si hay una recompensa por cada transacción que se realiza en el nodo de un particular. Del mismo modo, habrá que decidir si cada transacción que se haga conllevará un coste y, en caso afirmativo, si se haría con monedas comunes, con criptomonedas o con un token propio.

Vemos por tanto que, a pesar de que este proyecto demuestra que es técnicamente posible, blockchain es una tecnología que conlleva un gran cambio en cuanto a cómo asumimos que debe funcionar una aplicación. Antes de llevar esta idea a producción y permitir que usuarios finales puedan utilizarla, habrá que tener una hoja de ruta clara sobre hasta donde se quiere llegar y de cuánto debe abarcar la aplicación.

6.3 La viabilidad de blockchain

Debido a los sucesos que han ocurrido en el mundo de la tecnología y del blockchain concretamente desde que se empezó este proyecto, me gustaría matizar ciertos aspectos del proyecto a título personal.

A fecha de la escritura de esta sección (junio de 2021), estamos inmersos en la mayor escasez de semiconductores de la historia. Esto se debe a dos sucesos primordialmente:

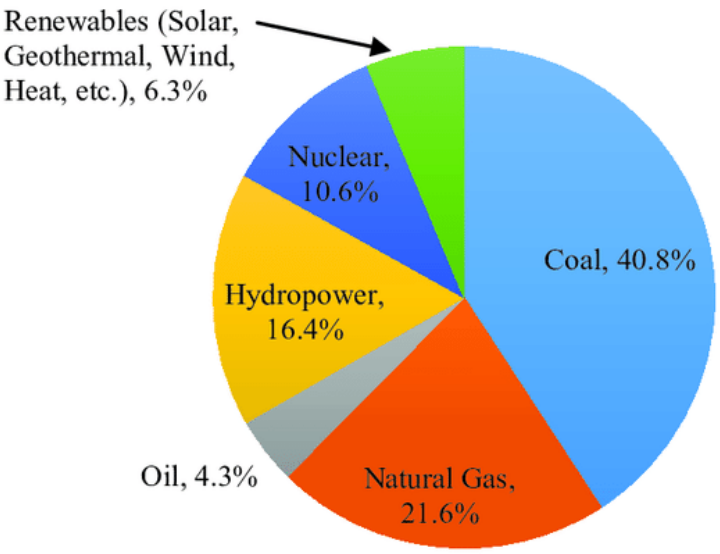

Figura 6.1: Principales fuentes de obtención de energía eléctrica en el mundo [16]

la pandemia de COVID-19 y, sobre todo, la constante subida de los precios de Bitcoin y Ethereum que han creado una segunda edad de oro de la minería de criptomonedas. Hay una enorme demanda de tarjetas gráficas, necesarias para la minería, y la poca oferta de chips que hay se está yendo a satisfacer dicha demanda.

El que un usuario final no pueda comprar una tarjeta gráfica a un precio que no esté inflado o que haya escasez de videoconsolas son simplemente la punta del iceberg de la explosión de las criptomonedas. Pero lo realmente peligroso son sus consecuencias para el medio ambiente.

Blockchain es, casi por definición, una tecnología muy poco eficiente. La cantidad de cálculos que hay que hacer para cada operación es enorme y, conforme pasa el tiempo y crece la blockchain, más difíciles son esos cálculos. Esto afecta especialmente el Bitcoin, donde, conforme más Bitcoins hay disponibles, se aumenta artificialmente la dificultad de los cálculos para que la recompensa sea más merecida.

Todos estos cálculos necesitan energía para realizarse y, en prácticamente todo el mundo, los combustibles fósiles siguen siendo la principal fuente para obtener energía eléctrica, como vemos en la figura 6.1 [16]. No podemos obviar que, por ejemplo, Elon Musk decidió dejar de minar y de permitir compras en sus empresas con Bitcoin apenas un mes después de anunciar un apoyo total a dicha criptomoneda. Dicha decisión se debió a la ineficiencia energética de blockchain [38].

En blockchain, siempre tendremos que hacer los mismos cálculos para cada transacción, independientemente de cuántas ventajas de la tecnología estemos usando. Es por ello que, a pesar de que ha quedado demostrado que es factible hacerlo, lo que he estudiado e investi-

gado durante estos meses me ha terminado poniendo en una posición un tanto contraria a la implementación de este proyecto. No hay ventajas suficientes que justifiquen la complejidad y el gasto, al menos mientras blockchain siga siendo ineficiente y un aumento de gasto energético pueda traer problemas en un futuro no tan lejano. Por tanto, si conseguimos incluir las ventajas que nos aporta blockchain en una tecnología más sencilla, habremos conseguido una mejor eficiencia. Tal vez ese debería ser el camino que sigan las investigaciones que partan de la misma problemática que este proyecto.

A

Anexo I: Manual de despliegue

En este capítulo detallaremos cómo desplegar la aplicación en el entorno local. Para ello, primero indicaremos los prerrequisitos que debe satisfacer el entorno donde hagamos el despliegue (esto es, el software a instalar) y, a continuación, los pasos que hay que seguir hasta tener la aplicación funcionando.

El script de despliegue está hecho para el sistema operativo Ubuntu en su versión 20.04.1, y es compatible con versiones posteriores. Por tanto, aunque se podría desplegar manualmente en otros sistemas operativos, las instrucciones se van a referir exclusivamente a cómo hacerlo en Ubuntu.

A.1 Prerrequisitos

Para poder utilizar la red de prueba de Hyperledger Fabric necesitaremos instalar:

- **Git**: la última versión disponible. Concretamente el proyecto utiliza la 2.25.1. Podemos hacer uso del comando `apt-get install git`
- **cURL**: de nuevo, la última versión disponible (en el proyecto se usó la 7.68.0), que podemos obtener mediante el comando `apt-get install curl`
- **Docker**: necesitaremos una versión superior a la 17.06.2-ce. En el proyecto se utilizó la versión 20.10.2. Para instalarlo, podemos seguir los siguientes pasos:

    1. Obtenemos el script de instalación: `curl -fsSL https://get.docker.com -o get-docker.sh`

2. Instalamos Docker: `sudo sh get-docker.sh`

   De esta manera nos aseguramos de instalar la última versión disponible. Además, necesitamos realizar tres pasos más:

3. Iniciamos el daemon de Docker: `sudo systemctl start docker`
4. Hacemos que el daemon se inicie al encenderse el sistema (este paso no es obligatorio pero sí recomendable, por comodidad): `sudo systemctl enable docker`
5. Añadimos nuestro usuario al grupo Docker, para poder ejecutar contenedores: `sudo usermod -a -G docker <username>`

- **Docker Compose**: cualquier versión superior a la 1.14.0. Nosotros utilizamos la versión 1.25.0. Para instalar esta en concreto, podemos seguir los siguientes pasos:

1. Obtenemos el script de instalación: `sudo curl -L "https://github.com/docker/compose/releases/download/1.25.0/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose` (Podemos sustituir el 1.25.0 en la URL por la versión que queramos instalar)
2. Hacemos el script ejecutable: `sudo chmod +x /usr/local/bin/docker-compose`

   Y ya estaría instalado. Para asegurarnos, podemos ejecutar el comando de versión: `docker-compose -version`

Con esto ya podríamos desplegar una red de prueba de Hyperledger, pero para nuestro prototipo nos faltan algunos detalles.

Primero, instalaremos Camunda. Descargaremos la última versión disponible de la Open Source Community Edition de la web oficial de Camunda (`camunda.com/download`). Nosotros usamos en concreto la versión 7.15.0. Descomprimimos el zip en el lugar que prefiramos. La ruta donde la hayamos descomprimido la necesitaremos a continuación. También necesitaremos la ruta del directorio donde hayamos descomprimido el código de este proyecto.

Abrimos el explorador de archivos, vamos a la carpeta home, pulsamos en el menú de hamburguesa y activamos la opción "show hidden files"(figura A.1). Hacemos doble click sobre el archivo .bashrc, y añadimos las líneas que aparecen en el listado A.1.

Listado A.1: Líneas a añadir en el fichero .bashrc

```
1 export CAMUNDA_HOST='[Directorio donde se haya descomprimido el .zip de
      Camunda]/start-camunda.sh'
2 export HLF_INSTALL_DIR=[Directorio donde se haya descomprimido el .zip
     del proyecto]/NeptunUS/hlf-test-network/
3 export PATH=[Directorio donde se haya descomprimido el .zip del
     proyecto]/NeptunUS/hlf-test-network/bin/:$PATH
```

Con esto, el script que despliega automáticamente la red de pruebas, instala Cicero en dicha red y despliega nuestro smart contract podrá ejecutarse sin problema.

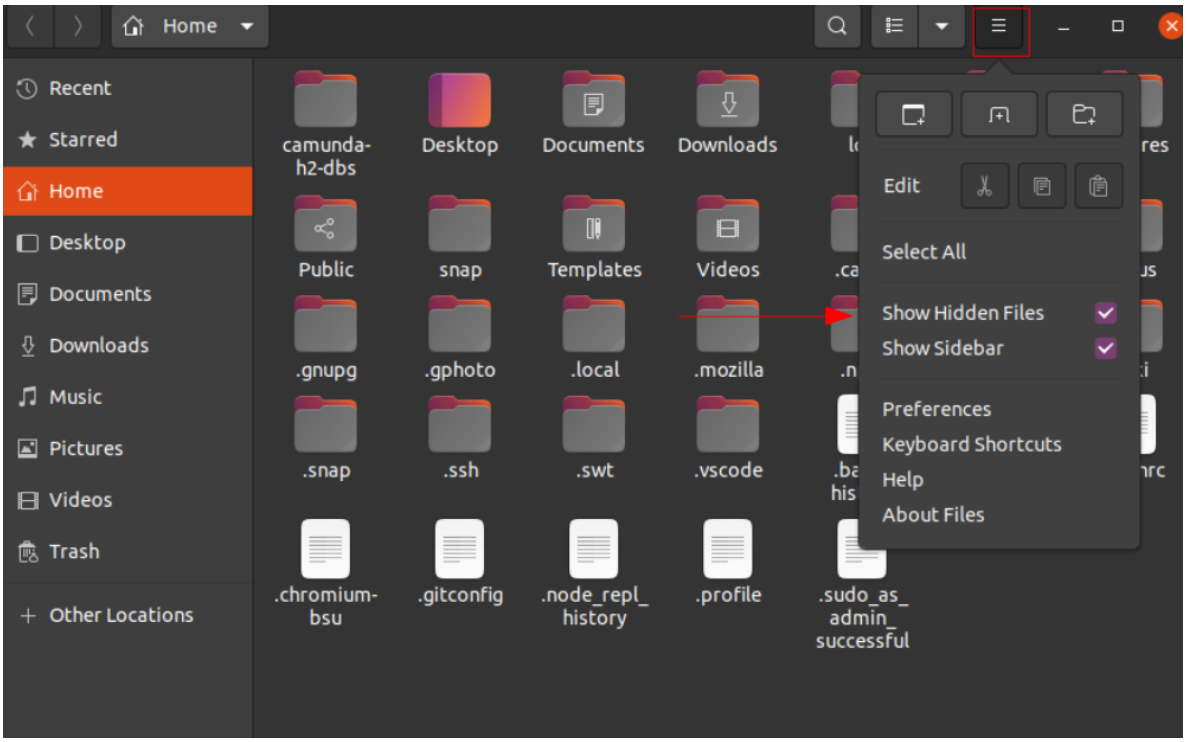

Figura A.1: Cómo mostrar los archivos ocultos

El último prerrequisito que necesitaremos será instalar el Modeler de Camunda para poder desplegar el proceso, que es el único paso que no podemos automatizar pero que sólo necesitaremos hacer una vez. Para ello, lo descargamos de la misma web que descargamos Camunda Platform (`camunda.com/download`) y descomprimimos el .zip donde queramos. Cuando queramos abrirlo, simplemente ejecutamos el archivo `camunda-modeler.sh`.

A.2 Despliegue

Para desplegar todo el prototipo, simplemente vamos a la carpeta NeptunUS y ejecutamos el archivo `neptunus.sh`. Si no queremos desplegar el total de la aplicación sino sólo ciertas partes, tenemos otros tres scripts en la misma carpeta raíz:

- `cicero-network-up.sh` para desplegar la blockchain y el host de Camunda. Espera un argumento, la ruta a un archivo .md que contenga el contrato a desplegar. Un ejemplo de este archivo se encuentra en `Neptunus/hlf-cicero-contract/santander-contract.md`.
- `workers-up.sh` para inicializar los workers de Camunda.
- `frontend-start.sh` para desplegar la aplicación de frontend en React.

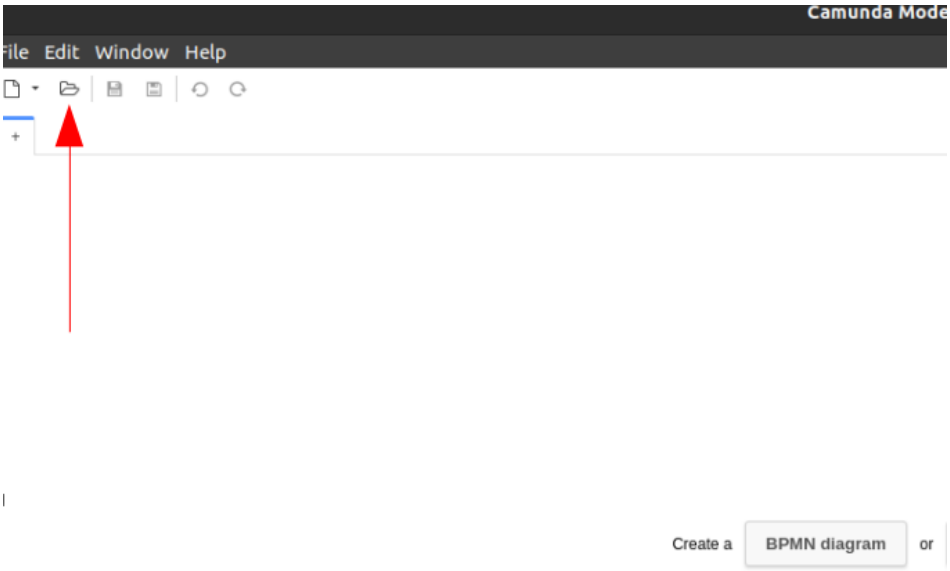

Figura A.2: Botón para abrir un nuevo diagrama en Camunda Modeler

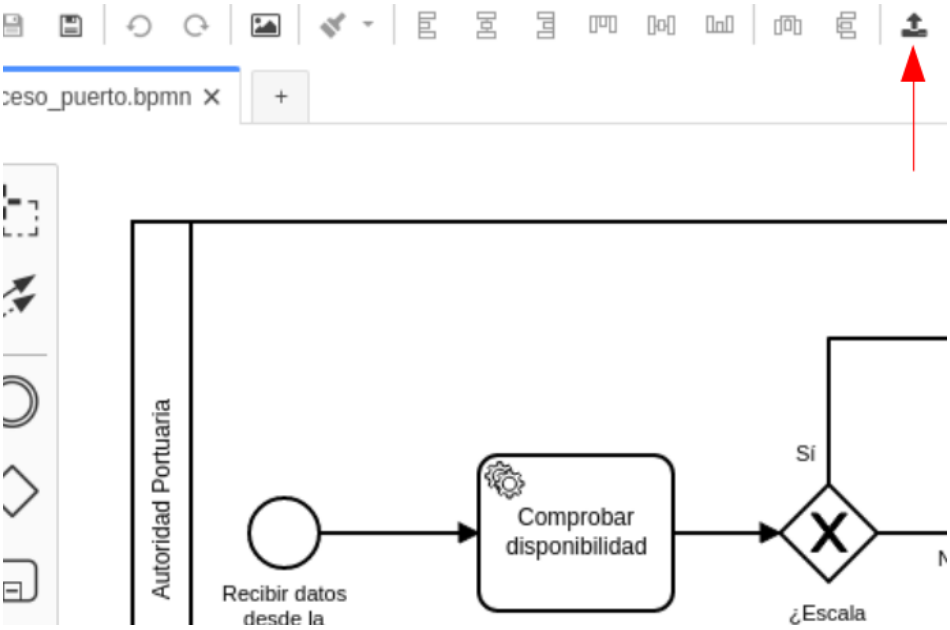

Figura A.3: Botón para desplegar el proceso en Camunda Modeler

Si es la primera vez que desplegamos el contrato, necesitaremos ejecutar un paso más. Abrimos el Modeler de Camunda y pulsamos sobre el archivo de la carpeta (Open New Diagram, figura A.2). Vamos a la ruta NeptunUS/camunda y hacemos doble click sobre `diagrama_proceso_puerto.bpmn`.

A continuación, hacemos clic sobre el botón con una flecha hacia arriba (Deploy Current Diagram, figura A.3) y, sin cambiar ninguna opción, hacemos click sobre el botón azul, Deploy. Con esto ya tendremos nuestro proceso funcionando en el host de Camunda.

B

Anexo II: Manual de usuario

En este capítulo, mostraremos cómo solicitar el registro de una escala en la aplicación desde el punto de vista del usuario. Para ello, detallaremos paso por paso cómo realizar una consulta, así como las posibles excepciones que nos podemos encontrar en el proceso. Se incluyen además varias imágenes del proceso.

Al introducirnos en la URL de la aplicación (por defecto, `localhost:3000`), se mostrará una página como la que nos muestra la figura B.1. En dicha figura hemos señalado dos ítems interesantes: el primero, señalado con 1 en la figura, es un selector que cambiará el formulario según la zona en la que queramos atracar. El segundo, marcado con 2 en la misma figura, es un mensaje de error que aparecerá hasta que la fecha de salida introducida sea posterior a la de llegada. Además, cabe señalar que el arqueo se puede introducir manualmente o se puede calcular usando los tres campos y presionando el botón Calcular.

Para enviar el formulario, presionaremos sobre el botón que aparece al final de la misma, Enviar. Como vemos en la figura B.2, mientras haya algún error o algún campo obligatorio no esté seleccionado, el botón aparecerá deshabilitado, en gris. En el momento en el que se pueda enviar, aparecerá en negro, con sombreado azul al poner nuestro cursor sobre el mismo.

Una vez que hayamos enviado el formulario, aparecerá una animación de carga en la página que vemos en la figura B.3. En función del resultado que obtengamos, la siguiente página variará. Si hay escala disponible veremos la página que aparece en la figura B.4, si no la hay iremos a una página como la que se muestra en la figura B.5 y si ha ocurrido algún error, aparecerá la página que se puede ver en la figura B.6.

Si queremos volver a empezar, simplemente refrescamos la página.

Figura B.1: Pagina de inicio de la aplicación

Figura B.2: Comportamiento del botón de Enviar

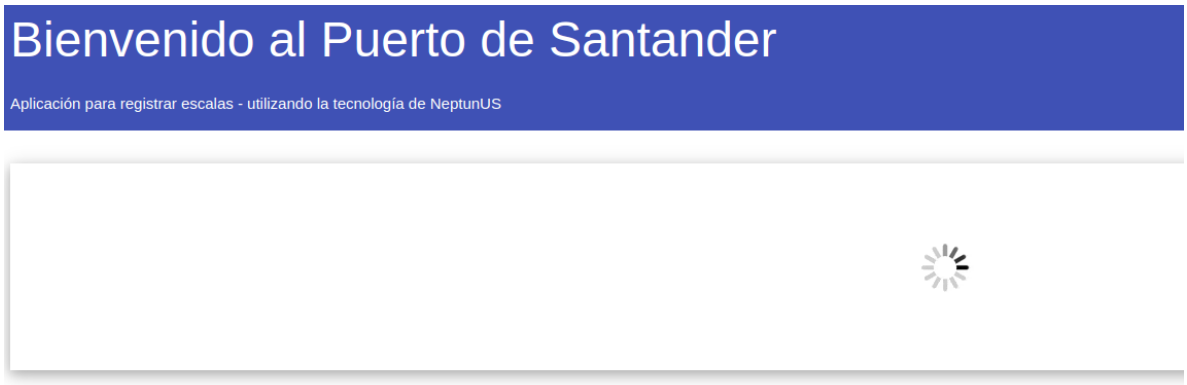

Figura B.3: Animación de carga

Figura B.4: Resultado: éxito

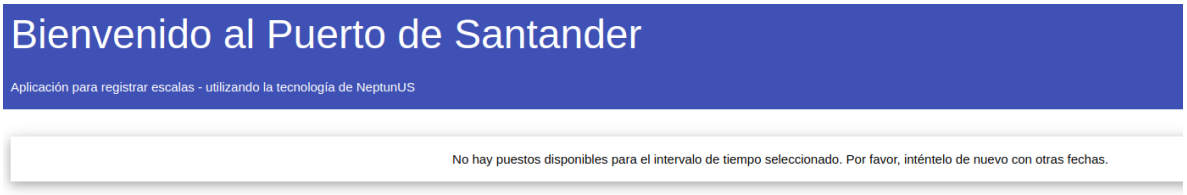

Figura B.5: Resultado: no hay escala disponible

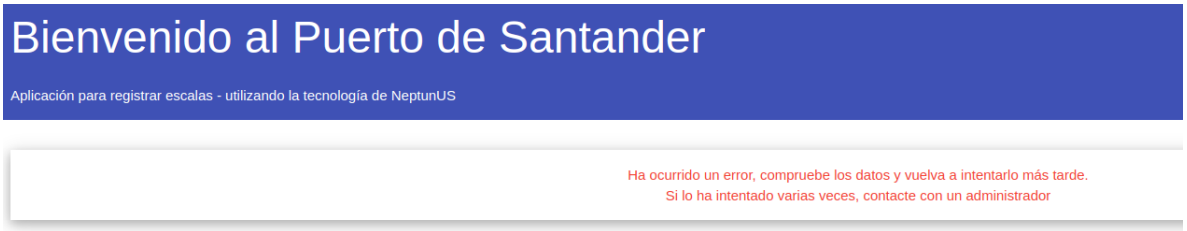

Figura B.6: Resultado: error

Bibliografía

[1] Documentación de Cicero. docs.accordproject.org.

[2] Documentación de Solidity. docs.soliditylang.org.

[3] Página de inicio de Activiti. activiti.org.

[4] Página de inicio de Angular. angular.io.

[5] Página de inicio de Bonitasoft. bonitasoft.com.

[6] Página de inicio de Camunda. camunda.com.

[7] Página de inicio de Ethereum. ethereum.org.

[8] Página de inicio de Gestión de Proyectos. gestionproyectos.us.es.

[9] Página de inicio de HyperLedger. hyperledger.org.

[10] Página de inicio de OpenLaw. openlaw.io.

[11] Página de inicio de React.js. reactjs.org.

[12] Página de inicio de RedMine. redmine.org.

[13] Página de inicio de Vue.js. vuejs.org.

[14] Boletín Oficial del Estado, núm. 221: Convenio internacional de 23 de junio de 1969, sobre Arqueo de Buques, hecho en Londres, Septiembre 1982. boe.es.

[15] REAL DECRETO 1185/2006, de 16 de octubre, por el que se aprueba el Reglamento por el que se regulan las radiocomunicaciones marítimas a bordo de los buques civiles españoles., Noviembre 2006. mitma.gob.es.

[16] Characteristics Study of Photovoltaic Thermal System with Emphasis on Energy Efficiency, 2020. researchgate.net.

[17] La logística portuaria se enfoca en la digitalización y automatización de sus procesos, Agosto 2020. cadenadesuministro.es.

[18] El puerto de Huelva contrata una plataforma para digitalizar infraestructuras y procesos, Junio 2021. elmercantil.com.

[19] Las tecnologías se asientan cada vez más en el transporte y con ellas el sector portuario se digitaliza para ser 4.0, Abril 2021. cotransa.com.

[20] ¿Cuánto Cobra un Ingeniero de Software?, 2021. jobted.es.

[21] Microsoft Azure. Precios de Blockchain Service (versión preliminar). azure.microsoft.com.

[22] N. Calabrese. The Growing Popularity of Blockchain Technology, Marzo 2021. research.g2.com.

[23] U. W. Chohan. Non-Fungible Tokens: Blockchains, Scarcity, and Value. *Critical Blockchain Research Initiative (CBRI) Working Papers*, 2021.

[24] A. S. d. P. Crespo, L. I. C. García. Stampery Blockchain Timestamping Architecture (BTA)-Version 6. *arXiv preprint arXiv:1711.04709*, 2017.

[25] M. Crosby, N. Nachiappan, P. Pattanayak, S. Verma, V. Kalyanaraman. BlockChain Technology: Beyond Bitcoin. *Applied Innovation Review*, páginas 6–19, Junio 2016.

[26] Mozilla Devs. Hola, React. developer.mozilla.org.

[27] D. Dujak, D. Sajter. Blockchain applications in supply chain. En *SMART supply network*, páginas 21–46. Springer, 2019.

[28] L. Fernández Espinosa. Qué son los smart contracts o contratos inteligentes, Septiembre 2019. bbva.com.

[29] S. Ferretti, G. D'Angelo. On the ethereum blockchain structure: A complex networks theory perspective. *Concurrency and Computation: Practice and Experience*, 32(12):e5493, 2020.

[30] J. Frankenfield, E. Rasure. What is Hyperledger Fabric?, Marzo 2021. investopedia.com.

[31] L. García. Un poco de Camunda, Septiembre 2018. unpocodejava.com.

[32] R. V. García. Blockchain y criptomonedas. *La notaría*, (3):22–25, 2017.

[33] M. Gupta. *Blockchain for Dummies*. John Wiley & Sons, Inc, 3rd ibm limited edición, 2020.

[34] M. T. Gómez, B. Ramos. Modelo de negocio actual del puerto de Santander. 2020.

[35] V. Hernández-Santaolalla, A. Hermida. Más allá de la distopía tecnológica: videovigilancia y activismo en Black Mirror y Mr. Robot. *Servicio de Publicaciones de la Universidad Rey Juan Carlos*, páginas 53–65, Julio 2016.

[36] Hyperledger. GitHub - hyperledger/fabric-samples. github.com.

[37] A. Kulkarni. Blockchain: Applications in payments, Agosto 2017. europeanpaymentscouncil.eu.

[38] E. Musk. Tesla & Bitcoin, Mayo 2021. twitter.com.

[39] V. Muñoz Reyes. Hyperledger Fabric - ¿Qué es Hyperledger?, Mayo 2019. medium.com.

[40] M. Nofer, P. Gomber, O. Hinz, D. Schiereck. Blockchain. *Business & Information Systems Engineering*, 59(3):183–187, 2017.

[41] Autoridad portuaria del puerto de Santander. Tasas portuarias: Tasas del buque, 2015. puertosantander.es.

[42] Accord Project. Ergo Overview. docs.accordproject.org.

[43] Accord Project. GitHub - accordproject/hlf-cicero-contract. github.com.

[44] Accord Project. Introducing Types. docs.accordproject.org.

[45] T. K. Sharma. Blockchain & Role of P2P Network, Mayo 2021. blockchain-council.org.

[46] N. Szabo. Formalizing and securing relationships on public networks. *First monday*, 1997.

[47] A. J. Varela-Vaca, A. M. Reina Quintero. Smart Contract Languages: A Multivocal Mapping Study. *ACM Computing Surveys (CSUR)*, 54(1):1–38, 2021.

[48] L. Weiser. The IMO Number Explained, Septiembre 2019. pewtrusts.org.

[49] S. Wilkinson, T. Boshevski, J. Brandoff, V. Buterin. Storj a peer-to-peer cloud storage network. 2014.