



UNIVERSIDAD DE GRANADA

TRABAJO FIN DE GRADO
GRADO EN INGENIERÍA DE TECNOLOGÍAS DE
TELECOMUNICACIÓN

Diseño e implementación de un entorno de gestión forense basado en Blockchain e IPFS

Autor

Salvador Moreno Rodríguez

Director

Gabriel Maciá Fernández



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE
TELECOMUNICACIÓN

Granada, noviembre de 2019

Diseño e implementación de un entorno de gestión forense basado en Blockchain e IPFS

Salvador Moreno Rodríguez

Palabras clave: IPFS, cifrado, Blockchain, Ethereum, hash, Ganache, Truffle, Metamask, Smart Contract.

Resumen

Hoy en día debido a los rápidos y grandes avances de la tecnología se sufre un preocupante retraso en la seguridad y en la jurisprudencia de los sistemas desarrollados. No solamente se habla de ataques a servidores de empresas o hacia particulares sino también de lo que respecta a la privacidad de los individuos. Los usuarios regalan sus datos a compañías sin tener un mínimo control sobre el tratamiento de los mismos. Sumando todo esto a un uso inapropiado de la tecnología por parte de la sociedad como puede reflejarse en la aparición del bullying cibernético o en la suplantación de identidad.

Ante estas alarmantes situaciones, los ingenieros y el personal especializado en el sector trabajan para desarrollar tecnologías alternativas que permitan confiar en la distribución y en el correcto uso de los datos, en sistemas que impidan la intrusión en dispositivos, en protocolos de eliminación u ocultación de mensajes ofensivos, etc.

Así han surgido sistemas y protocolos como Blockchain o IPFS que pretenden alcanzar lo que se conoce como la Web3. Este es un nuevo concepto de Internet donde los usuarios no han de confiar en el comportamiento ético de los empresarios o en el correcto funcionamiento de los servidores, sino que se persigue la confianza sobre la tecnología en la que se sustentan las aplicaciones.

En este sentido aún queda mucho por hacer para otorgar al usuario final un completo sistema confiable, pero a día de hoy ya hay plataformas y aplicaciones que apuestan por la privacidad y la seguridad de las redes.

En lo que respecta a este proyecto, se propone un sistema para asegurar una cadena de custodia utilizando tecnologías de este tipo; es decir, se persigue impedir la alteración o eliminación de pruebas en un juicio confiando específicamente en los sistemas Blockchain y en el protocolo IPFS.

Design and implementation of a forensic management environment based on Blockchain

Salvador Moreno Rodríguez

Keywords: IPFS, encryption, Blockchain, Ethereum, hash, Ganache, Truffle, Metamask, Smart Contract.

Abstract

Nowadays, due to the rapid and great advances in technology, there is a disturbing delay in the security and jurisprudence of the developed systems. Not only is it a matter of attacking company servers or individuals, but also the privacy of individuals. The users give their data to companies without having a minimum control over the treatment of them. Adding all this to an inappropriate use of technology by society as can be reflected in the emergence of cyber bullying or phishing.

In regard to these alarming situations, engineers and personnel specialized in the sector are working to develop alternative technologies that allow to trust in the distribution and correct use of data, in systems that prevent intrusion into devices, in protocols for the elimination or concealment of offensive messages, etc.

This is how systems and protocols such as Blockchain or IPFS have arisen in order to achieve what is known as Web3. This is a new concept of the Internet where users do not have to trust in the ethical behaviour of businessmen or in the correct functioning of servers, but rather trust in the technology on which the applications are based.

In this sense there is still much to be done to give the end user a complete reliable system, but today there are platforms and applications that bet on the privacy and security of networks.

With regard to this project, a system is proposed to ensure a chain of custody using technologies of this type; that is, it seeks to prevent the alteration or elimination of evidence in a trial by relying specifically on Blockchain systems and the IPFS protocol.

Yo, **Salvador Moreno Rodríguez**, alumno de la titulación TITULACIÓN de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 20080934w, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Salvador Moreno Rodríguez

Granada a 15 de Noviembre de 2019 .

D. **Gabriel Maciá Fernández**, Profesor del Área de Ingeniería Telemática del Departamento de Teoría de la Señal, Telemática y Comunicaciones de la Universidad de Granada.

Informa:

Que el presente trabajo, titulado ***Diseño e implementación de un entorno de gestión forense basado en Blockchain e IPFS***, ha sido realizado bajo su supervisión por **Salvador Moreno Rodríguez**, y autoriza la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expide y firma el presente informe en Granada a 15 de Noviembre de 2019 .

El director:

Gabriel Maciá Fernández

Agradecimientos

Lo primero de todo agradecer a mi tutor Gabriel Maciá la oportunidad de trabajar con él, el tiempo empleado y la paciencia recibida. Seguidamente debo dar las gracias a mi familia por el apoyo incondicional, en especial a mi madre y a mi padre junto con mi hermana y mi hermano menor. No podría olvidar a mis mejores amigos de la infancia que me han acompañado hasta día de hoy como son: Victor, Josi, Alberto, Ismael, Rufino e Irene al igual que a Nacho y Toni desde que los conocí. Dar las gracias también a todos mis amigos del Barrio con los que he compartido momentos inmejorables. Cómo no hablar de las magníficas personas con las que he convivido durante la carrera y me han acompañado no solo en la biblioteca y en clase si no también en fiestas y en ese viajaado inolvidable a Tenerife: el gran Zapata, Pino, Ivanzu, Ramos, Domo, Reda, Yose, Pisha, Althammer, Placido, Manu, Salcedo, Bailón, Mase, Javi Guzmán, Irigaray, Javi, Quiles , Infantes y por su puesto a Glous.

Índice general

1. Introducción	1
1.1. Motivación y contexto del proyecto	1
1.2. Objetivos del proyecto y logros conseguidos	2
1.3. Estructura de la memoria	2
1.4. Contenidos teóricos para la comprensión del proyecto	3
1.4.1. Blockchain	3
1.4.2. Ethereum	7
1.4.3. Smart contract	8
1.4.4. Oráculo	10
1.4.5. Ganache	11
1.4.6. Truffle	11
1.4.7. Metamask	11
1.4.8. IPFS	12
1.4.9. Git	12
1.4.10. BitTorrent	13
1.4.11. Tablas Hash Distribuidas (DHT)	14
1.4.12. Sistema de archivos certificados de forma segura (SFS)	15
1.4.13. Multihash	16
1.4.14. BitSwap	17
1.4.15. Merkle DAG	18
1.4.16. IPNS	19
2. Planificación y costes	21
2.1. Costes	23
3. Análisis del problema	25
3.1. Especificación de requisitos	25
3.2. Análisis	26
4. Diseño	29
4.1. Diseño global del sistema	29
4.2. Especificaciones IPFS	35

5. Implementación	39
5.1. Javascript	39
5.2. Código del almacenamiento de la evidencia	40
5.3. Código de comprobación y descarga de la evidencia	44
6. Evaluación y pruebas	47
7. Conclusiones	53
7.1. Valoración personal	53
Bibliografía	56
Glosario de siglas	57
A. Manual de usuario	59
A.1. Requisitos	59
A.2. Instalación de paquetes necesarios	59
A.3. Pasos a seguir	60

Índice de figuras

1.1.	Gasto de GAS de los distintos tipos de operaciones	9
1.2.	Estructura que sigue la identificación de los nodos en Kademlia	14
1.3.	Ejemplo ruta de autocertificación de un archivo en SFS	16
1.4.	Ejemplo lista de ficheros dentro de un directorio en IPFS . .	18
2.1.	Diagrama de Gantt	21
4.1.	Esquema general del proyecto	30
4.2.	Comportamiento de la interfaz gráfica.	31
4.3.	Primeras interacciones en Ganache	32
4.4.	Funciones del Oráculo	33
4.5.	Ultimas interacciones en Ganache.	34
4.6.	Función del fichero COMPROBACIÓN.	35
4.7.	Inicio IPFS	36
4.8.	Ventana de bienvenida de IPFS	36
4.9.	Inicialización del daemon	37
4.10.	Interfaz web de IPFS	38
6.1.	Mensaje enviado de @Prueba1Tfg a @Prueba2T	47
6.2.	Página cuyo contenido se quiere denunciar	48
6.3.	Funcionamiento de la interfaz <i>web</i>	48
6.4.	Confirmación del pago en <i>Metamask</i>	49
6.5.	Notificación de éxito del almacenamiento	49
6.6.	Bloque de <i>Ganache</i> donde se almacena la evidencia	50
6.7.	Eliminación de pruebas	51
6.8.	Respuesta del fichero <i>COMPROBACIÓN.js</i>	52
6.9.	Ampliación del <i>hash</i> devuelto	52
A.1.	Configurando Ganache	61
A.2.	Configurando Ganache	62
A.3.	Configurando Ganache	63
A.4.	Importación de la cuenta en Metamask	64
A.5.	Dirección de los contratos	65
A.6.	Interfaz web	66

A.7. Confirmación de pago en Metamask	67
A.8. Saldo de las cuentas cuando se ejecuta el pago	67
A.9. Bloques minados	68
A.10. Bloque que contiene los datos de la función <i>solicitarDatos()</i> .	68
A.11. Bloque que contiene los datos de la función <i>guardarSolicitud()</i>	68
A.12. Bloque que contiene el pago del Cliente al Oráculo	69

Índice de tablas

1.1. Estructura del bloque.	4
1.2. Estructura de la cabecera.	4
2.1. Mano de obra.	23
2.2. Material.	23
2.3. Costes indirectos.	24
2.4. Coste total del proyecto.	24

Listados de código

1.1. Ejemplo acceso al contenido de un fichero de IPFS	18
1.2. Formato IPNS	19
1.3. Ejemplo DNS en IPFS	19
4.1. Inicializar el perfil de IPFS	35
4.2. Mostrar página de bienvenida de IPFS	36
4.3. Inicializar el daemon	36
5.1. Ejemplo callback hell	39
5.2. Ejemplo Promise	40
5.3. Ejemplo async/await	40
5.4. Comienzo del Cliente y el Oráculo	41
5.5. Función mediante la cual el Cliente solicita los datos al Smart- contract	41
5.6. Evento que escucha el Oráculo	41
5.7. Activación del evento y función general del Oráculo	41
5.8. Función del Oráculo que conlleva más tiempo de ejecución . .	42
5.9. Subida de datos y devolución del enlace IPFS	43
5.10. Función mediante la cual el Oráculo envía los datos al Smart- contract	44
5.11. Evento que escucha el Cliente	44
5.12. Introducción de los datos en el fichero COMPROBACIÓN.js	44
5.13. Contenido función main()	45
6.1. Inicialización del daemon de IPFS	51
6.2. Inicialización del daemon de IPFS	51

Capítulo 1

Introducción

1.1. Motivación y contexto del proyecto

Actualmente la sociedad se enfrenta a innumerables delitos informáticos más allá de la intrusión en equipos ajenos o el espionaje en las comunicaciones. Cada día son más los casos de difusión de imágenes sin consentimiento a través de las redes sociales, acoso a los usuarios, *bullying* infantil, suplantación de identidad, etc.

Si bien el personal cualificado en el sector trabaja duramente para implantar mecanismos que impidan o dificulten la realización de estos actos ilícitos, el mal uso de la tecnología por parte de la sociedad hace que sea muy difícil controlar todas estas situaciones.

Cuando los afectados de estos abusos denuncian un delito, son los peritos informáticos los encargados de registrar las pruebas y custodiarlas hasta la celebración del juicio. Este procedimiento es lo que se conoce como cadena de custodia cuyo objetivo es asegurar que las pruebas presentadas ante el tribunal coinciden con las recogidas en primera instancia.

Aunque pueda parecer una actividad sencilla, los peritos informáticos en muchas ocasiones se enfrentan a serios problemas a la hora de obtener estas evidencias. Dichas dificultades se producen debido a que el contenido denunciado puede ser borrado o manipulado por parte de los propietarios de la empresa que desarrolla la aplicación mediante la cual se comete el delito, el usuario denunciado puede intentar eliminar su rastro en la red o utilizar un perfil anónimo, etc.

No solo se presentan trabas cuando se pretende obtener las evidencias, sino también a la hora de almacenarlas ya que puede pasar mucho tiempo desde la presentación de la denuncia hasta la celebración del juicio, lo que puede suponer una ventaja para posibles implicados que quieran eliminar la información comprometida de los discos duros o de las plataformas donde se aloja.

Si se desea conocer más en cuanto al proceso de recolecta de evidencias

se puede consultar en [1] .

Como alternativa, en este proyecto se plantea un sistema que permita a las instituciones policiales descargar las pruebas expuestas en una página web desde el momento de su denuncia, guardarlas en una plataforma segura y asegurar su veracidad ante los tribunales.

Todo gracias a la invención de nuevos sistemas descentralizados basados en la distribución de la información a través de distintos nodos en la red. Más concretamente de la tecnología *Blockchain* y del protocolo *Interplanetary File System (IPFS)*. Además, se hablará de la plataforma de código abierto *Ethereum* y de las aplicaciones utilizadas en el desarrollo del proyecto.

1.2. Objetivos del proyecto y logros conseguidos

En este proyecto se ha modificado un sistema existente que permite a los usuarios almacenar una evidencia en una base de datos segura mediante el uso de la tecnología *Blockchain*.

Se busca un método para acreditar que la información almacenada en primera instancia corresponde con la que se pueda presentar en un juicio a posteriori, para ello, se ha añadido un enlace a **IPFS** donde se pueden descargar dichos datos en el caso de que la evidencia devuelta por el sistema anterior no coincida con la del informe presentado.

1.3. Estructura de la memoria

Este trabajo consta de distintas secciones como puede verse en el índice del proyecto.

- En la Sección 1.4 se explica el funcionamiento y fundamento teórico de las principales tecnologías utilizadas para facilitar la comprensión de los demás apartados.
- Posteriormente en la Sección 2 se plasma el tiempo invertido en la creación del proyecto junto con los gastos económicos que conllevaría implementarlo.
- Para entender mejor el objetivo perseguido, en la Sección 3 se especifican las funcionalidades conseguidas junto con las soluciones elegidas ante los retos que se han presentado en su desarrollo.
- En la Sección 4 se da una explicación detallada de la infraestructura construida y como interactúan las partes que la forman para lograr los objetivos perseguidos.

- Seguidamente en la Sección 5 se desarrollan las especificaciones llevadas a cabo en cada uno de los bloques de diseño.
- Para terminar se muestra un apartado donde se justifica el correcto funcionamiento del proyecto en la Sección 6 y otras secciones donde se pueden encontrar las conclusiones, bibliografía, glosario y un manual para que el usuario lo implemente en su equipo.

1.4. Contenidos teóricos para la comprensión del proyecto

1.4.1. Blockchain

Blockchain [2] como se conoce actualmente salió a la luz con el lanzamiento de la criptomoneda *Bitcoin* [3] en 2008. Se trata de un algoritmo de consenso que permite el almacenamiento de documentos de manera segura mediante la participación de entidades confiables llamados **mineros**. Se considera una base de datos distribuida por el hecho de que los datos se guardan en todos los nodos que forman la red y también descentralizada debido a que no existe una infraestructura central que los interconecte.

La base de datos está compuesta por bloques de información que se almacenan encadenándose unos con otros (de ahí el nombre de la tecnología) a través de la criptografía. Para la mejor comprensión del funcionamiento de esta innovadora tecnología, a continuación, se explican los conceptos que permiten un sistema de intercambio de criptomonedas como Bitcoin (BTC), aunque tenga muchas más aplicaciones.

En este método la cantidad de BTCs que una cuenta posee se calcula en función de las transacciones recogidas en la cadena de bloques. Por lo tanto, si una cuenta **A** recibe **5** BTCs en el bloque 1, y envía **2** BTCs a **B** en el bloque 2, en el bloque 3 no podrá enviar **4** BTCs a **C** ya que **A** no dispone de ese saldo porque según el cómputo global de la cadena: **A** tiene en su poder **3** BTCs. En los bloques además de las transacciones, se recogen otros datos necesarios para el correcto funcionamiento del sistema: una cabecera, el tamaño en *bytes* del bloque y el número de transacciones que contiene dicho bloque como puede verse en la Tabla 1.1.

En la **cabecera** es donde se introducen las pautas que regulan la inserción de los bloques como puede verse en la Tabla 1.2:

Se ha de recalcar la aparición del campo **hash**. Este es una herramienta muy utilizada a día de hoy en las comunicaciones cifradas. Se trata de un algoritmo matemático que transforma un conjunto de datos en una serie de caracteres con una longitud fija. La ventaja reside en que el *hash* obtenido en este algoritmo cambia en función de los datos de entrada. Concretamente en *Bitcoin* se utiliza el protocolo de encriptación *SHA-256* del que se hablará

Tamaño	Campo	Descripción
4 bytes	Tamaño de Bloque	El tamaño del bloque en bytes
80 bytes	Cabecera de Bloque	La componen varios campos
1-9 bytes	Contador de Transacción	Cuántas transacciones hay en el bloque
Variable	Transacciones	Las transacciones registradas en este bloque

Tabla 1.1: Estructura del bloque.

Tamaño	Campo	Descripción
4 bytes	Versión	Un número de versión para seguir las actualizaciones de software y protocolo
32 bytes	Hash del Bloque Anterior	Una referencia al hash del bloque anterior (padre) en la cadena
32 bytes	Raíz Merkle	Un hash de la raíz del árbol de merkle de las transacciones de este bloque
4 bytes	Sello de tiempo	El tiempo de creación aproximada de este bloque en formato UNIX
4 bytes	Objetivo de Dificultad	El objetivo de dificultad del algoritmo de prueba de trabajo para este bloque
4 bytes	Nonce	Un contador usado para el algoritmo de prueba de trabajo

Tabla 1.2: Estructura de la cabecera.

más adelante cuando se describa el concepto de *multihash* en la Sección 1.4.13.

La raíz *Merkle* es otro parámetro que, como se indica en la Tabla 1.2, es el *hash* resultante de combinar los *hashes* de todas las transacciones recogidas en el bloque en forma de árbol.

También aparecen los campos de: sello de tiempo, objetivo de dificultad y *nonce* que son los parámetros necesarios en la **minería** explicada inmediatamente a continuación.

La **minería** es el proceso por el cual se consigue insertar los bloques de datos en la cadena. Se trata de una actividad que depende del algoritmo de consenso de los mineros siendo estos los nodos encargados de insertar y validar los bloques testificando que las transacciones son verídicas. Se suponen entidades confiables pues, para participar en la red, se requiere de un tipo de esfuerzo en función del algoritmo utilizado.

Existen muchos algoritmos de consenso como son; Proof of Work (PoW), Proof of Stake (PoS), Proof of Authority (PoA) entre los más conocidos. En esta sección se hablará principalmente del algoritmo PoW ya que es el más utilizado por ser el primero que se inventó. Aún así, al final de este apartado se comentan las características principales de los otros dos y cómo afrontan el problema que supone la minería con PoW. Como se mencionó en el párrafo anterior, si se quiere participar en la minería se tiene que pagar un precio para participar. En el caso de PoW el esfuerzo consiste en agotar recursos propios en electricidad y equipos con el objetivo de obtener un *hash* válido. Como se verá seguidamente, conseguir un *hash* válido es una meta muy costosa si la dificultad introducida en el algoritmo es muy alta, por lo que se necesita un alto poder de computación. Ese es el motivo principal por el que los mineros que trabajan con PoW son fabricados con tarjetas gráficas conectadas en paralelo o son equipos especializados llamados *ASICS* (Para más información consultar [4]) que requieren un elevado consumo de corriente para su funcionamiento y una inversión económica importante para la construcción del minero.

El *hash* depende de los parámetros de la cabecera del bloque en cuestión como entrada, de manera que, si el campo **dificultad** establece que el *hash* para ser válido ha de contener 12 ceros al principio, hasta que el minero no lo consiga, no se insertará el bloque. Este parámetro se reajusta cada “X” número de bloques (cambia en función de la versión) con el objetivo de que se tarde aproximadamente un intervalo de tiempo fijo dependiendo de la *blockchain* (en *Bitcoin* son 10 minutos) que también está introducido en la cabecera. Por lo tanto, una vez añadido el *hash* de las transacciones, el único valor mutable que contiene la cabecera es el **nonce**. A efectos prácticos este es el parámetro que se tiene que encontrar para validar un bloque. Por eso, los mineros van cambiando este contador de manera aleatoria hasta que consigan el objetivo descrito en el campo **dificultad**. Cuando un minero tiene la suerte, que realmente depende de la potencia de minado que posea

para realizar cálculos por unidad de tiempo (*hashes* por segundo), y consiga dar con un *nonce* que cumpla las condiciones del bloque, este inserta una transacción hacia su cuenta recibiendo *Bitcoin* como recompensa. Para finalizar, envía su bloque a los demás nodos que comprobarán que el *nonce* es válido y si es así se añadirá a la cadena.

Es importante entender por qué se usan los *hashes* en este algoritmo. Si la cadena estuviera compuesta por 5 bloques y un minero malicioso **D** quisiera beneficiarse introduciendo una transacción falsa en el bloque 2 (donde **A** enviaba 2 BTCs a **B**) en la cual intentase que **A** le enviase los 2 BTCs a su cuenta, el *hash* del campo **raíz de Merkle** de la cabecera del bloque 2 cambiaría provocando una modificación en el *hash* de la cabecera del bloque 2, que a su vez afectaría al *hash* del campo **hash del bloque anterior** de la cabecera del bloque 3 y así sucesivamente hasta el último bloque. He aquí la gran ventaja en cuanto a la seguridad que aporta la tecnología *blockchain*.

Según lo que se comentó al principio del apartado, la *blockchain* se trata de una base de datos distribuida puesto que todos los mineros almacenan la cadena de bloques completa, esto conlleva a que, a pesar del coste computacional que supone y del incentivo de la recompensa económica tanto por realizar correctamente la actividad como por las comisiones cobradas en las transacciones, si algún nodo intentase introducir un bloque defectuoso, necesitaría como mínimo el 51 % de la minería en el instante de la acción; es decir, necesitaría poseer 51 de cada 100 mineros para conseguir su objetivo. Esto se debe a que como se ha comentado anteriormente, antes de introducir un bloque válido en la cadena, el minero debe enviar ese bloque a los demás para que la confirmen, por lo tanto, si la mayoría de nodos no contienen la misma cadena, esta se dará por errónea y se continuará con el proceso de inserción del mismo bloque.

Un aspecto crucial en esta tecnología es la **sincronía**, ya que todos los mineros trabajan para insertar el siguiente bloque de la cadena en un periodo de tiempo establecido por el sistema. Cuando un minero consigue insertar un bloque en la base de datos, todos los demás se descargan la cadena completa y comienzan a trabajar en el siguiente bloque. A lo que se quiere llegar es que es necesario limitar el tamaño de los bloques (concretamente en *Bitcoin* se limitan a 3500 transacciones) ya que no todos los mineros tienen la misma velocidad de descarga y esto podría afectar a la sincronía en cuestión. Aquí es donde surge el problema de la escalabilidad de la *blockchain*. Ante esto han aparecido alternativas donde en vez de almacenar un volumen de datos limitado en los bloques, se almacenen enlaces a bases de datos sin restricciones de capacidad como IPFS.

Para acabar con esta sección se debe explicar el principal problema del algoritmo de consenso PoW. Con el aumento de la participación en la minería, la dificultad para insertar un bloque ha aumentado hasta el nivel de tener que formar **pools** de minería. Estos son agrupaciones de mineros que prestan su potencia de cómputo al servicio de la comunidad con el fin de

aumentar la probabilidad de que uno de ellos encuentre el *hash* válido para insertar el siguiente bloque. De esta manera se reparten las ganancias entre los que forman la comunidad en función de la potencia servida por cada uno de ellos. Esto conlleva un gasto de flujo energético a nivel global enorme únicamente dedicado a la minería, por lo que es prácticamente inviable, o muy poco eficiente, mantener un sistema escalable que utilice esta tecnología.

Como alternativa, otras *blockchains* como *Ethereum* han empezado a utilizar el algoritmo de consenso **PoS**. En este método, el esfuerzo soportado en la minería en vez de agotar recursos energéticos, es la necesidad de tener **saldo congelado** en la cuenta del minero; es decir, tener acumuladas una cantidad de *criptomonedas* de manera que, cuanto más tiempo y más *criptomonedas* se tengan sin utilizar, más probabilidad hay de que el minero inserte el siguiente bloque. Una motivación para que el minero realice bien su trabajo, es la desaparición de ese saldo en el caso de que intente insertar un bloque erróneo. El principal problema es que al solo necesitar una cuenta para participar, un usuario que contenga un saldo congelado que suponga un poder de minado mayor o igual al 51 %, no sería posible alterar la situación como en PoW ya que no existe la posibilidad de que aparezca un usuario con mayor poder de minado.

Otro algoritmo de consenso como se mencionó anteriormente es el caso de **PoA**. En este método (utilizado también en *Ethereum*), es la **reputación** de los mineros lo que se pone en juego como medida de transparencia. Estos mineros deben ser votados por otros mineros previamente autorizados lo que supone una centralización en el sistema ya que no todo el mundo puede participar. Como ventaja respecto a los anteriores es que no consume los recursos propios de PoW y no incentiva al estancamiento del poder adquisitivo como en PoS, pero supone una amenaza ante la privacidad de los participantes y una facilidad a la hora de censurar la información almacenada en los bloques.

1.4.2. Ethereum

Ethereum [5] es una plataforma de código libre desarrollada por *Vitalik Buterin* (coproductor de la revista: “*Bitcoin Magazine*”) cuyo objetivo es facilitar el desarrollo de aplicaciones descentralizadas mediante el uso de la tecnología *blockchain*.

Esta plataforma utiliza su propia criptomoneda llamada Ether (ETH) cuyo valor económico depende (como en *Bitcoin*) de la oferta y la demanda de la misma. Además, se caracteriza por la creación de contratos inteligentes autoejecutables conocidos como **smart contracts**. Estos contratos permiten el despliegue de aplicaciones sobre un sistema distribuido, lo que se traduce en una dificultad pronunciada a la hora de censurar el servicio y en la promesa de un tiempo de funcionamiento confiable.

A los mineros de *Ethereum*, cada vez que insertan un bloque en la ca-

dena se les recompensa con un número de ETHs por defecto. Además, son remunerados por parte de los usuarios en función del coste computacional que conlleve la ejecución de sus transacciones o de sus *smart contracts* en forma de **gas** (en vez de ETHs). El *gas* es una unidad que sirve para medir el gasto computacional de la red, por lo que realmente no tiene ningún valor y no se puede almacenar en una cuenta o en un monedero. Lo que se pretende con ello es diferenciar el valor económico de la criptomoneda con el valor computacional que conlleva el sistema. Si el coste computacional dependiese de un activo tan volátil como el ETH, significaría que cuando el valor de la criptomoneda fuese muy bajo, el coste que conlleva la minería también disminuiría, pero esto no es así ya que depende de factores como la electricidad o el deterioro de los equipos. Como ejemplo, el coste por realizar una simple transacción es de 21000 *gas*, sin importar los ETHs que se envíen de una cuenta a otra. El coste computacional total que requiere la ejecución de un contrato lo devuelve el sistema y lo denominan “*Gas Used*”.

En las transacciones y en los contratos inteligentes, se debe indicar la cantidad de *Gigaweis* (Subdivisión del ETH, 1 *Gwei* = 0.000000001 ETH) que el usuario está dispuesto a pagar a los mineros por cada unidad de *gas*, lo que se conoce como “*Gas Price*”. Aunque anteriormente se ha dicho que el *gas* en sí no tiene valor, este campo se utiliza para regular el pago a los mineros en función del valor del ETH en el mercado. Si el *gas* tuviese un valor económico inmutable (un número fijo de *Gweis*) y el valor del ETH en el mercado fuese muy bajo, si se quisiera realizar una transacción que conlleva un gasto fijo de 21000 *gas*, se estaría pagando tan poco a los mineros que nos les saldría rentable tramitar la transacción. Así mismo, si el precio del ETH fuese muy alto, se estaría pagando demasiado.

El sistema a la hora de hacer una transacción o lanzar un contrato, devuelve al usuario un valor de “*Gas Used*” utilizado junto con un “*Gas Price*” orientativo. A raíz de ahí, el usuario decide si aumentar o disminuir el “*Gas Price*” en función de la velocidad con la que desee que se realice su transacción. Esto se debe a que los mineros escogerán primeramente las transacciones y contratos con los que saquen un mayor beneficio y dejarán para más tarde los demás.

Esta plataforma cuenta con varias *blockchains* de prueba como son: “*Ropsten*”, “*Rinkeby*”, “*Kovan*” y “*Goerly*”, que utilizan dinero ficticio y distintos algoritmos de consenso con el fin de proporcionar a los desarrolladores un espacio de trabajo sin riesgos. No obstante, la cadena principal donde se guardan todos los contratos y transacciones que conlleven un gasto real recibe el nombre de “*Main*”.

1.4.3. Smart contract

Los *smart contracts* son *scripts* que se autoejecutan cuando se cumplen unas condiciones preestablecidas. Están escritos comúnmente en **Solidity** [6]

(lenguaje de alto nivel basado en *C++*, *Javascript* y *Python*) y almacenados en la cadena de bloques. Su objetivo es implementar un estado de seguridad mayor que con los contratos tradicionales reduciendo sus costes y sus tiempos de tramitación.

Para posibilitar su desarrollo en diferentes ordenadores y mejorar así la portabilidad del sistema, *Ethereum* creó su propia máquina virtual denominada: *Ethereum Virtual Machine (EVM)* que interpreta el **bytecode** resultante de la compilación de los *smart contracts*. Un *smart contract* puede estar escrito en lenguaje de alto nivel (*Solidity*) o en ensamblador (*opcode*). El **opcode** es un lenguaje más cercano al *bytecode* (lenguaje máquina que interpreta la EVM) que muestra las operaciones que se deben ejecutar para el funcionamiento del contrato. A partir de ahí, se realiza el cálculo del “*Gas Used*” (mencionado en el apartado 1.4.2) ya que cada operación tiene asociado un consumo de *gas* fijo en función de su coste computacional. El gasto de cada operación se muestra en la Figura 1.1 obtenida del *yellow paper* [7] de *Ethereum*.

Name	Value	Description*
G_{zero}	0	Nothing paid for operations of the set W_{zero} .
G_{base}	2	Amount of gas to pay for operations of the set W_{base} .
$G_{verylow}$	3	Amount of gas to pay for operations of the set $W_{verylow}$.
G_{low}	5	Amount of gas to pay for operations of the set W_{low} .
G_{mid}	8	Amount of gas to pay for operations of the set W_{mid} .
G_{high}	10	Amount of gas to pay for operations of the set W_{high} .
$G_{extcode}$	700	Amount of gas to pay for operations of the set $W_{extcode}$.
$G_{balance}$	400	Amount of gas to pay for a BALANCE operation.
G_{sload}	200	Paid for a SLOAD operation.
$G_{jumpdest}$	1	Paid for a JUMPDEST operation.
G_{sset}	20000	Paid for an SSTORE operation when the storage value is set to non-zero from zero.
G_{sreset}	5000	Paid for an SSTORE operation when the storage value's zeroness remains unchanged or is set to zero.
R_{sclear}	15000	Refund given (added into refund counter) when the storage value is set to zero from non-zero.
$R_{selfdestruct}$	24000	Refund given (added into refund counter) for self-destructing an account.
$G_{selfdestruct}$	5000	Amount of gas to pay for a SELFDESTRUCT operation.
G_{create}	32000	Paid for a CREATE operation.
$G_{codeDeposit}$	200	Paid per byte for a CREATE operation to succeed in placing code into state.
G_{call}	700	Paid for a CALL operation.
$G_{callvalue}$	9000	Paid for a non-zero value transfer as part of the CALL operation.
$G_{callstipend}$	2300	A stipend for the called contract subtracted from $G_{callvalue}$ for a non-zero value transfer.
$G_{newaccount}$	25000	Paid for a CALL or SELFDESTRUCT operation which creates an account.
G_{exp}	10	Partial payment for an EXP operation.
$G_{expbyte}$	50	Partial payment when multiplied by $\lceil \log_{256}(exponent) \rceil$ for the EXP operation.
G_{memory}	3	Paid for every additional word when expanding memory.
$G_{txcreate}$	32000	Paid by all contract-creating transactions after the <i>Homestead</i> transition.
$G_{txdatazero}$	4	Paid for every zero byte of data or code for a transaction.
$G_{txdatanonzero}$	68	Paid for every non-zero byte of data or code for a transaction.
$G_{transaction}$	21000	Paid for every transaction.
G_{log}	375	Partial payment for a LOG operation.
$G_{logdata}$	8	Paid for each byte in a LOG operation's data.
$G_{logtopic}$	375	Paid for each topic of a LOG operation.
G_{sha3}	30	Paid for each SHA3 operation.
$G_{sha3word}$	6	Paid for each word (rounded up) for input data to a SHA3 operation.
G_{copy}	3	Partial payment for *COPY operations, multiplied by words copied, rounded up.
$G_{blockhash}$	20	Payment for BLOCKHASH operation.
$G_{quaddivisor}$	20	The quadratic coefficient of the input sizes of the exponentiation-over-modulo precompiled contract.

Figura 1.1: Gasto de GAS de los distintos tipos de operaciones

Con el fin de evitar bucles infinitos y prevenir errores humanos en la creación de los *smart contracts* que puedan agotar el saldo de una cuenta, cada vez que se quiere insertar un contrato en la *blockchain* se ha de indicar un valor de “*Gas Limit*”. Este valor limita la cantidad de *gas* que se puede consumir y lo fija el usuario en función del “*Gas Used*” obtenido en la compilación del contrato. Así pues, el valor del “*Gas Limit*” ha de ser mayor que el del “*Gas Used*”; de lo contrario, cuando se alcance este valor, la transacción fallará y además el usuario tendrá que pagar las operaciones tramitadas hasta ese momento por el minero.

Los *smart contracts* están compuestos por su contenido y dos claves públicas; una que representa al propio contrato (necesaria para su invocación) y otra que identifica a su creador. Para su inserción en la *blockchain*, el creador envía el *bytecode* del contrato en una transacción para que los mineros lo inserten en un bloque. A continuación, el *smart contract* solo se podrá activar cuando un usuario lo invoque. Una vez haya sido activado por un usuario, tanto otros usuarios como otros contratos podrán recurrir a él. Según la finalidad del contrato, este puede estar formado por funciones públicas o privadas. Si son públicas, todos los participantes y contratos de la red podrán utilizarlas siempre y cuando paguen a los mineros el coste que conlleve la operación y en el supuesto de que sean privadas solo las cuentas elegidas dentro del contrato podrán usarlas. Cuando el contrato se almacena en la *blockchain* ya no se puede modificar, de hecho, solo se podrá eliminar de la cadena si contiene una función específica llamada *SELFDESTRUCT*, en caso contrario, el contrato quedará latente a la espera de ser utilizado. Esta inmutabilidad de los contratos presenta el inconveniente de que si el *smart contract* dependiese de un valor no almacenado en la *blockchain* no podría funcionar ya que no habría ningún contrato dentro de la cadena al que se pudiera recurrir. Por ejemplo, si un *smart contract* se programase para que se transfirieran todos los ETHs de una cuenta a otra cuando el valor del ETH superase una cierta cantidad de dólares, se necesitaría de una fuente de información externa a la *blockchain* que indicase dicho valor. Para estos casos se crearon los oráculos explicados en 1.4.4.

1.4.4. Oráculo

Los **oráculos** son todas las aplicaciones, páginas web o librerías externas a la *blockchain* que ofrecen una confianza plena sobre la veracidad de los datos requeridos por los *smart contracts*. En el ejemplo donde un usuario mediante un contrato quería enviar todos sus ETHs cuando su valor superase una cierta cantidad de dólares, se necesitaba de alguna fuente externa a la cadena de bloques que aporte esa información. Esta fuente a la que se acude es el oráculo. Los oráculos llegan a acuerdos con la *blockchain* prometiendo fidelidad de los datos requeridos y disponibilidad continuada. Aún así, si un atacante quisiera interferir en la funcionalidad de un *smart contract*, el

oráculo sería el punto débil del sistema ya que no dispone de la seguridad que aporta la tecnología *blockchain*. Para solventar esta posibilidad, en el código del contrato se suele acceder a múltiples oráculos que aporten la misma información, pero pertenecientes a distintas empresas o plataformas. De esta manera se comparan los datos de diferentes puntos de internet por si alguno de ellos ha sido alterado por entidades maliciosas.

1.4.5. Ganache

Con el objetivo de facilitar el aprendizaje de la tecnología *blockchain*, han surgido múltiples plataformas para la creación de una cadena de bloques. **Ganache** [8] es un *framework* que permite su implementación a nivel local proporcionando 10 cuentas con saldo ficticio y donde el único minero de la red es el desarrollador. Lo que se persigue es evitar la espera en la inserción de los bloques que conlleva la minería y el gasto económico de las transacciones requerido por el sistema para su funcionamiento.

1.4.6. Truffle

Truffle [9] es un entorno de desarrollo que permite implementar *smart contracts* de forma segura. Cuenta con un compilador de *Solidity*, un entorno de programación para *testing* y permite la migración de los contratos para insertarlos en distintas *blockchains* como *Ganache*. El inconveniente de **Truffle** si se quieren migrar los *smart contracts* a *blockchains* de prueba como *Rinkeby*, *Ropsten*, *Kovan* o *Goerly* es que necesitas un cliente de *Ethereum* completo para su inserción; es decir, tienes que minar para estas cadenas. Con el fin de evitar al usuario la configuración de un nodo completo para formar parte de la red y agotar los recursos que conlleva la minería, surgió un *plugin* para los navegadores *Chrome* y *Firefox* llamado **Metamask** que permite conectar al usuario con estas *blockchains* sin la necesidad de que monten un minero.

1.4.7. Metamask

Metamask [10] es una extensión para *Firefox* y los buscadores basados en *Chromium* como *Google Chrome*, *Opera* o *Brave*. Este *plugin* de código abierto sirve de puente entre *Ethereum* y los navegadores anteriores, conectándose a nodos de *Ropsten*, *Rinkeby*, *Kovan*, *Goerly* y al *Main*, o incluso a *blockchain* locales como *Ganache*. **Metamask** permite crear cuentas nuevas para cada *blockchain* o importar otras ya existentes pertenecientes a los usuarios. Gracias a esta plataforma, se pueden realizar transacciones sin la necesidad de gestionar las claves públicas y privadas asociadas a cada cuenta. Además, cuenta con un modo que protege al usuario ante páginas maliciosas utilizadas para *phishing* (Para conocer más sobre esta técnica consultar <https://github.com/MetaMask/eth-phishing-detect>).

1.4.8. IPFS

IPFS [11] es un sistema de archivos distribuidos *peer-to-peer*; es decir, está formado por nodos que se comportan como iguales entre sí, actuando como clientes y servidores, permitiendo el intercambio directo de la información. Estos nodos se identifican mediante el *hash* de sus claves públicas y se interconectan para formar una red de almacenamiento en bloques cuyo direccionamiento se realiza a través hipervínculos cifrados.

Para facilitar la comprensión del funcionamiento de esta tecnología a continuación se detallan los conceptos clave de los protocolos en los que se basa: **Git** (Sección 1.4.9), **BitTorrent** (Sección 1.4.10), **Distributed Hash Table (DHT)** (Sección 1.4.11) y **Self-Certified File System (SFS)** (Sección 1.4.12).

Después se hablará sobre:

- **Multihash** (Sección 1.4.13): encriptación que utiliza el protocolo.
- **BitSwap** (Sección 1.4.14): el protocolo de intercambio de bloques de información que utiliza IPFS.
- **Merkle DAG** (Sección 1.4.15): la manera en la que estructura los archivos.
- **Interplanetary Name System (IPNS)** (Sección 1.4.16): el algoritmo que usa para resolver problemas de inmutabilidad.

1.4.9. Git

Git [12] es un sistema de control de versiones distribuido creado por *Linus Torvalds* en 2005 (fundador del *Kernel* de *Linux*) que organiza los datos en un sistema de ficheros en árbol y permite introducir cambios en los archivos de manera sencilla. La principal diferencia respecto a anteriores *Version Control System (VCS)* es la forma en la que **Git** trata los cambios que se realizan sobre un fichero. Mientras que otros sistemas almacenan las modificaciones en un archivo, **Git** realiza instantáneas del fichero cada vez que se confirma o se guarda el estado del proyecto. La ventaja es que, si no se realizan cambios, el protocolo devuelve un enlace al archivo anterior en vez de almacenarlo de nuevo. Esta manera de trabajar permite tener varias bifurcaciones de una rama principal en la que se pueden realizar cambios sin modificar el contenido de la misma, hasta que el usuario desee fusionar el contenido de todas las ramas con el fin acoplar la información de una forma eficaz ya que solo se insertan los cambios que haya entre ellas. Los ficheros son representados como objetos inmutables denominados; *blob*, los directorios como; *tree* y los cambios realizados como; *commit*.

1.4.10. BitTorrent

BitTorrent [13] es un protocolo de intercambio de archivos *peer-to-peer* cuya ventaja reside en la velocidad de transferencia de datos debido a la participación conjunta de redes de pares no confiables. El sistema se compone de:

- *Peers*: son todos los usuarios que participan en la red.
- *Leechers*: son los clientes que están descargando un archivo de la red y no lo poseen aún. También se les denomina así a los nodos que únicamente se dedican a descargar archivos sin compartirlos con otros.
- *Seeders*: son los usuarios que poseen el archivo completo.
- *Trackers*: son servidores especiales que permiten localizar a los pares.
- *Swarm*: son agrupaciones de nodos que poseen un archivo completo entre ellos.

Para reducir la latencia de descarga, **BitTorrent** típicamente divide la información en paquetes de 16 *kilobytes* con el fin de que sea más sencillo para un usuario buscar dichos paquetes en diferentes nodos. Este protocolo obliga a que cuando se inicia la descarga de un fichero, hasta que no se obtienen todos los paquetes que lo forman, no se pueda empezar a descargar otro. Ante el problema de que no se consiga un archivo completo, los nodos buscan y descargan primeramente los paquetes menos comunes dejando para más tarde los que abundan en el enjambre. Cuando una semilla carga por primera vez un archivo, al no conocer los paquetes más raros, esta selecciona trozos al azar hasta que se completa la subida de datos. Podría ocurrir que un par que está enviando un paquete tenga un reducido ancho de banda de subida por lo que, para no ralentizar la descarga, el nodo que requiere el archivo envía mensajes de búsqueda de los demás paquetes a todos los pares del enjambre. Una vez que el nodo lo recibe, este envía mensajes de cancelación de búsqueda del mismo con el objetivo de reducir la sobrecarga en la red.

El éxito que ha tenido este protocolo se debe al *ojo por ojo* que implementa, método que incentiva a los pares a cargar los paquetes y no solo a descargarlos, cooperando así a que disminuya la latencia del sistema a la hora de obtener cualquier archivo. Se trata de que los nodos que gastan sus recursos para subir los datos a la red se asocien entre ellos de manera que puedan bajar información con una mayor velocidad. Igualmente, los pares que solo se dedican a obtener los archivos de otros, se ven “asfixiados” ya que les resulta más difícil encontrar nodos que precisen dichos datos.

1.4.11. Tablas Hash Distribuidas (DHT)

Las **DHTs** son estructuras de datos que se utilizan para distribuir y encontrar información de forma fluida en sistemas de redes distribuidos. Para ello usan dos tipos de identificadores; una **clave**, que sirve para dar con el nodo que almacena los datos deseados y un **valor** que representa al archivo requerido. Lo que distingue a las DHTs con tablas *hash* ordinarias es que las búsquedas y el almacenamiento de la información se distribuye entre pares que pueden salir y entrar de la red libremente. **IPFS** utiliza una mezcla de tres protocolos basados en tablas hash distribuidas:

- **Kademlia DHT** [14]: Se caracteriza por minimizar los mensajes de configuración que necesitan los nodos para conectarse entre sí. Estos mensajes se realizan en segundo plano informando de la distancia entre nodos. Basándose en este factor se consigue que el enrutado elegido sea óptimo. Las ventajas de este sistema son; el uso de la función *XOR* para el cálculo de la distancia entre los nodos de la red y el envío de consultas asíncronas con el fin de evitar retrasos en las comunicaciones por nodos caídos. En *Kademlia* los nodos se distribuyen en una estructura en árbol (como puede verse en la Figura 1.2 sacada de [14]) de manera que cada uno mantiene contacto con al menos otro de su misma rama.

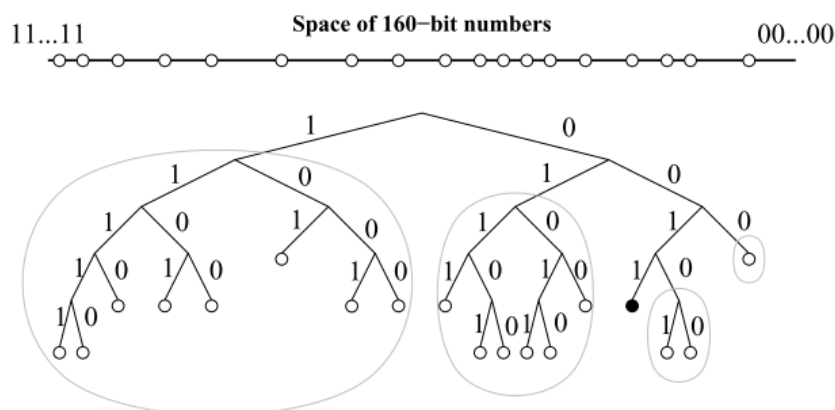


Figura 1.2: Estructura que sigue la identificación de los nodos en Kademlia

Cada nodo posee una tabla donde se indica la dirección IP, el puerto y el nombre que lo identifica junto con los conjuntos <clave, valor> de los próximos a él. El nombre de cada nodo es representado con 160 bits ya que se obtiene encriptando su clave pública con el protocolo SHA-1.

- **Coral DSHT** [15]: Es una extensión del protocolo anterior que busca

disminuir el ancho de banda y la sobrecarga en los nodos que contienen información muy demandada. Esto se consigue introduciendo clústeres alrededor de dichos nodos que se organizan y comunican entre sí de manera automática para evitar el requerimiento de servidores lejanos o excesivamente solicitados. Distributed Sloopy Hash Table (DSHT) incorpora tablas dedicadas en función del tiempo de *ping* entre nodos. Esto supone una mejora respecto a *Kademlia* ya que sus tablas DHT solo informan de la distancia entre los puntos de la red y no sobre la latencia de los enlaces.

- ***S/Kademlia DHT*** [16]: Esta propuesta surge con la necesidad de dar soporte ante ataques como:
 - *Sybil*: Consiste en insertar diversos nodos con el objetivo de hacerse con una parte de la red.
 - *Eclipse*: Se introduce una gran cantidad de nodos alrededor de uno específico para asegurarse que el enrutamiento se efectúa por al menos uno malicioso, ocultando así el nodo objetivo.
 - *Rutas adversas*: Con el fin de anular un nodo de la tabla de enrutamiento de los demás que forman parte de la red, el atacante envía información de encaminamiento alternativa aprovechando que *Kademlia* elimina los puntos que no responden o que no enrutan ningún paquete.

Para solventar estos problemas el protocolo implementa algunas mejoras como son:

- Aumentar el coste de recursos a la hora de insertar un nodo en la red. Para ello introduce una firma supervisada junto con un sistema de cripto puzles que exigen un consumo elevado de CPU y de ancho de banda.
- Imposibilitar la libre elección del identificador del nodo que influye en la posición del mismo.
- Insertar un algoritmo de búsqueda de rutas inconexas para evitar que existan caminos a los que no se puedan acceder desde cualquier punto de la red.

1.4.12. Sistema de archivos certificados de forma segura (SFS)

SFS [17] es un sistema de archivos descentralizado que aporta una sólida seguridad en redes no confiables ya que utiliza criptografía de claves públicas en cada servidor. En cualquier otro sistema esto supondría una gestión de claves para poder conectarse a dichos servidores, sin embargo, este protocolo da la posibilidad a los usuarios que quieran formar parte de la red a utilizar

técnicas distintas de certificación de claves de los nodos con el fin de tener una mayor versatilidad.

$$\overbrace{\text{/sfs/sfs.lcs.mit.edu}}^{\text{Location}} \overbrace{\text{:vefvsv5wd4hz9isc3rb2x648ish742h}}^{\text{HostID (specifies public key)}} \overbrace{\text{/pub/links/sfscvs}}^{\text{path on remote server}}$$

Figura 1.3: Ejemplo ruta de autocertificación de un archivo en SFS

En la Figura 1.3 (obtenida en [17]) se muestra una ruta de autocertificación de un archivo que el cliente ha de utilizar para acceder a él. En la primera parte, el cliente debe introducir la localización del servidor ya sea con el dominio *DNS* o la dirección *IP*. A continuación, se tiene que insertar la identificación del nodo (**HostID**) para asegurar un canal seguro hacia dicho servidor. El *HostID* es un *hash* criptográfico encriptado con el protocolo *SHA-1* que se obtiene como salida de; la clave pública del servidor, la localización del mismo e información adicional que el nodo quiera introducir. De manera que, si el cliente quiere comprobar la autenticidad del enlace, le puede solicitar la clave pública y la información adicional al nodo y realizar la misma codificación. Este método permite a cualquier usuario participar en la red si lo desea ya que solo necesita; una dirección de Internet, una clave pública (que puede generar como él prefiera), determinar el *HostID* y ejecutar el software de SFS para referenciar cualquier archivo.

1.4.13. Multihash

Antes de pasar a hablar de los protocolos que ha creado IPFS basándose en lo anterior, se ha de describir el algoritmo de encriptación que sigue. IPFS utiliza lo que se denomina **multihash**. Se trata de un algoritmo de encriptación donde se describe: el protocolo elegido, la longitud en *bytes*, y el *hash* resultante: **<protocolo><longitud><hash>**

Un ejemplo real de lo anterior podría ser:

122041dd7b6443542e75701aa98a0c235951a28a0d851b11564d20022ab11d2589a8

Donde cada dígito es un número en hexadecimal. Dos dígitos en hexadecimal corresponden a un byte.

<12><20><41dd7b6443542e75701aa98a0c235951a28a0d851b11564d20022ab11d2589a8>

De tal forma que los dos primeros dígitos: 12 (o 0x12) son 0001 0010 en binario. Esta primera cifra informa del protocolo elegido para la encriptación que según la tabla **multihash** (se puede consultar en <https://github.com/multiformats/multicodec/blob/master/table.csv>) es **SHA2-256**.

Los siguientes dos dígitos se refieren a la longitud en *bytes* del *hash*. Por lo que 0x20 en hexadecimal corresponde a 32 en decimal. De ahí que tengamos un *hash* de 32 *bytes* (256 *bits*).

El resto de dígitos es el *hash* resultante de aplicar el algoritmo de encriptación.

En IPFS es muy común encontrar *hashes* que comiencen por “**Qm**”, esto se debe a que utiliza la estructura anterior, pero en vez de estar codificada en hexadecimal, está codificada en Base58.

En realidad, IPFS permite cualquier protocolo de encriptación siempre y cuando siga el formato *multihash*.

1.4.14. BitSwap

BitSwap [11] es el protocolo basado en *BitTorrent* 1.4.10 que IPFS usa para el intercambio de información entre nodos con la ayuda de las DHTs para el enrutamiento. Se consigue gracias al envío de dos tipos de listas:

- *Want-List*: En ella se exponen los paquetes de los archivos deseados.
- *Have-List*: Aquí se listan los paquetes de los archivos almacenados.

Cuando un nodo recibe una *Want-List* comprueba si tiene alguno de esos ficheros en su *Have-List* y los envía en caso afirmativo. En el caso contrario, reenvía esa *Want-List* a sus nodos vecinos hasta que se encuentre la información requerida. Al contrario de *BitTorrent*, **BitSwap** no limita la descarga de bloques de información continuados hasta que se complete un mismo archivo, si no que permite a los nodos adquirir los paquetes que deseen, aunque pertenezcan a archivos distintos. Otra diferencia respecto a *BitTorrent*, es que cuando un nodo no posee información que le interese al resto, este se dedica a buscar los paquetes deseados por los demás nodos recurriendo a sus *Want-List*. El objetivo es incentivar la prestación de datos por parte de los usuarios para que no los cataloguen como *leechers* (explicados en 1.4.10). La manera que tiene **BitSwap** para castigar a los pares que no prestan servicios a sus compañeros se rige por una función probabilística que depende de:

$$r = \frac{bytes_{sent}}{bytes_{recv} + 1} \quad (1.1)$$

Siendo r en la Ecuación 1.1 el parámetro que se usa para describir la actividad prestada por parte de un nodo en función de la recibida por el que requiere los paquetes. Así, la probabilidad de que el usuario se los envíe se rige por la Ecuación 1.2:

$$P(send/r) = 1 - \frac{1}{1 + e^{6-3r}} \quad (1.2)$$

De manera que si r es muy pequeña (significando que se ha recibido muchos paquetes anteriores del nodo demandante) la probabilidad de que el nodo poseedor le envíe los datos que requiere al solicitante es muy alta.

1.4.15. Merkle DAG

Merkle DAG [11] es una generalización de la estructura de datos de *Git* donde el contenido se identifica por su *hash* y se representa en forma de árbol. IPFS divide los archivos en bloques, de manera que, si el archivo tiene un volumen superior a 256kB, este se fragmenta en paquetes de ese tamaño. A cada bloque se le asigna un *hash* correspondiente y al estar estructurados y enlazados en forma de árbol, el usuario final podrá comprobar la totalidad del archivo deseado si fusiona todos los paquetes y dan como resultado el *hash* raíz. Los directorios se organizan en carpetas y subcarpetas identificadas con un *hash* **inmutable**. Cuando introducimos un archivo en una carpeta, el sistema lo referencia con su *hash* de contenido; es decir, no importa el nombre que el usuario le otorgue al fichero ya que, si subimos el mismo archivo con dos nombres distintos, en esa carpeta solo saldrá uno de ellos debido a que el *hash* resultante de encriptar los bytes del contenido con el protocolo *SHA256*, es el mismo. IPFS usa el mismo sistema que *Git* para evitar la sobrecarga de la red; en vez de almacenar dos archivos iguales, en el segundo intento se guardará el enlace del *hash* que referencia al archivo almacenado en primera instancia. A pesar de esto, IPFS permite a los nodos que quieran asegurar la permanencia de un objeto, descargarlo en su disco local.

En la Figura 1.4 sacada de [11] se muestra cómo se pueden listar los ficheros dentro de un directorio de IPFS.

En el supuesto de que se quisiese acceder al contenido del fichero “script” en IPFS se puede realizar a través de su *hash* de contenido o con la ruta completa desde el directorio raíz como puede verse en el Listado de código 1.1.

```
> ipfs ls /XLZ1625Jjn7SubMDgEyeaynFuR84ginqvzb
XLYkgq61DYaQ8NhkcqyU7rLcnSa7dSHQ16x 189458 less
XLHBNmRQ5sJJrdMPuu48pzeyTtRo39tNDR5 19441 script
XLF4hwVHsVuZ78FZK6fozf8Jj9WEURMbCX4 5286 template

<object multihash> <object size> <link name>
```

Figura 1.4: Ejemplo lista de ficheros dentro de un directorio en IPFS

```
1 $: ipfs cat /ipfs/XLHBNmRQ5sJJrdMPuu48pzeyTtRo39tNDR5
2 $: ipfs cat /ipfs/XLZ1625Jjn7SubMDgEyeaynFuR84ginqvzb/script
```

Listado de código 1.1: Ejemplo acceso al contenido de un fichero de IPFS

1.4.16. IPNS

Como se ha mencionado en el apartado anterior, IPFS realiza un *hash* al contenido de los archivos, referenciándolos y direccionándolos en función al mismo. Este protocolo presenta la ventaja de poder almacenarlos una sola vez en el sistema, comprobar la autenticidad de los enlaces o incluso ofrece una resistencia ante la manipulación del contenido gracias a la encriptación.

Respecto a esta última aplicación surge el problema de que los contenidos son inmutables. Este hecho podría acarrear dificultades a la hora de actualizar la información de un archivo si se deseara. Como solución, IPFS ofrece un sistema basado en SFS llamado: **IPNS** [11] .

Cuando se quiere actualizar el contenido de un archivo inmutable, IPFS asigna un espacio de nombres a cada nodo aprovechando que están identificados por el *hash* de sus claves públicas (***NodeId***= *hash (clave pública)*). El formato de ficheros de un mismo nodo se puede ver en 1.2

```
1 /ipns/<NodeId>/<fichero_firmado_clave_privada>
2 /ipns/XLF2ipQ4jD3UdeX5xp1KBgeHRhemUtaA8Vm/<fichero1 >
3 /ipns/XLF2ipQ4jD3UdeX5xp1KBgeHRhemUtaA8Vm/<fichero2 >
4 /ipns/XLF2ipQ4jD3UdeX5xp1KBgeHRhemUtaA8Vm/<fichero3 >
```

Listado de código 1.2: Formato IPNS

La idea es que haya un mismo *hash* (***NodeId***) con el que se pueda acceder a distintos ficheros inmutables. Cada fichero estará cifrado con la clave privada del nodo que lo haya publicado de manera que, para descifrarlo, haga falta la clave pública del mismo.

De esta manera un usuario puede asegurarse de que el fichero ha sido publicado por el nodo correspondiente ya que su *NodeId* debe corresponder al *hash* de su clave pública (necesaria para descifrar el fichero).

Así, aunque los ficheros sean inmutables, se puede actualizar la información deseada si se crea un nuevo fichero y se accede al *NodeId* del nodo que originó el primer archivo.

Para resumir, el *NodeId* de cada nodo es un puntero que apunta a ficheros inmutables, de manera que se puede acceder a la información actualizada si se introduce por ejemplo la dirección del último fichero.

Este es un sistema que se sigue optimizando a día de hoy. Como alternativa al *NodeId* se ha implementado un sistema basado en Domain Name System (DNS) que en vez de dar un nombre de dominio a una dirección IP, se lo otorga a un *NodeId*. El formato para acceder a un fichero quedaría como en el Listado de código 1.3

```
1 /ipns/example.com/<fichero1>
```

Listado de código 1.3: Ejemplo DNS en IPFS

Capítulo 2

Planificación y costes

En este proyecto de fin de grado se han llevado a cabo distintas tareas propuestas por el tutor para facilitar la elaboración del mismo. A continuación se muestra un diagrama de *Gantt* que permite visualizar de forma secuencial los bloques principales (color negro) junto con las actividades realizadas en cada uno (color azul) .

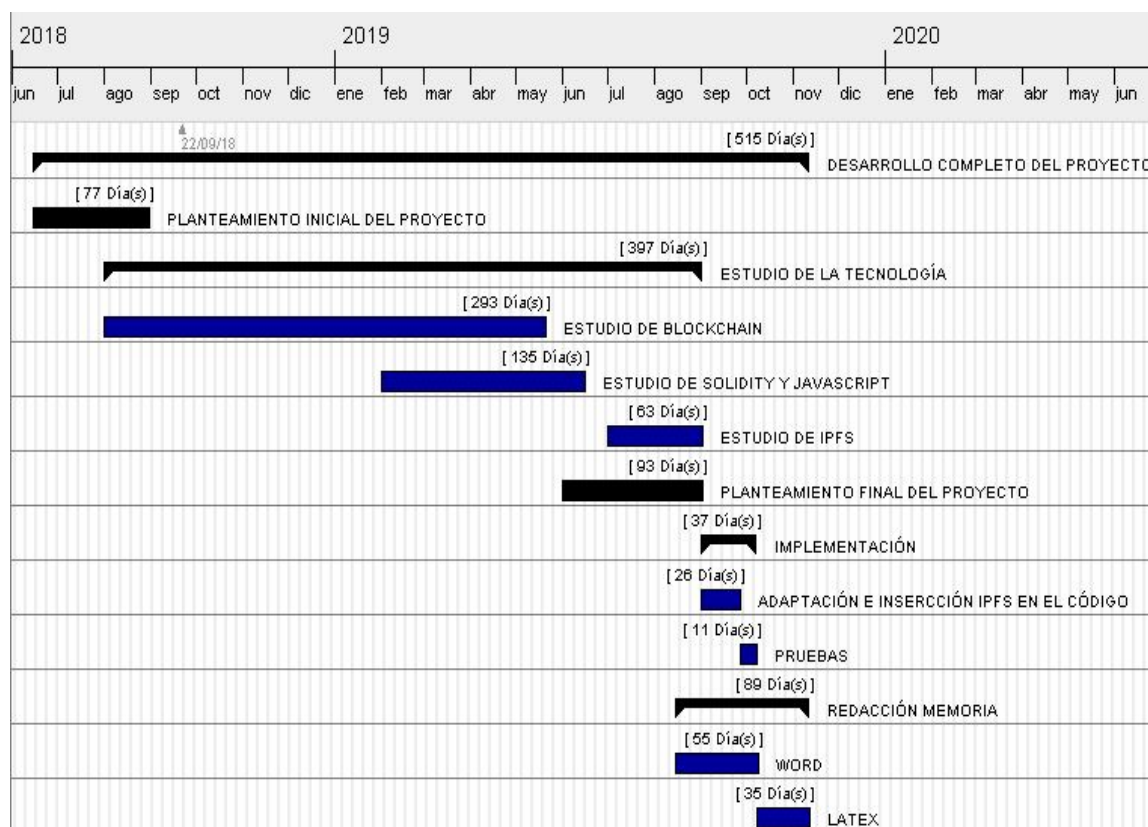


Figura 2.1: Diagrama de Gantt

Como puede observarse en la Figura 2.1 las tareas realizadas han sido:

- **Planteamiento inicial del proyecto:** En este periodo se propuso realizar una herramienta con la que se analizase una vulnerabilidad el código de los *smart contracts*. Con este objetivo se procedió al estudio de la tecnología.
- **Estudio de *Blockchain*:** Ha sido la mayor parte de este trabajo debido a la complejidad e innovación que le caracteriza. Principalmente se centró en entender el funcionamiento de las cadenas de bloques, los algoritmos de consenso, la minería, la interacción con los mineros, el funcionamiento de *Bitcoin* y de *Ethereum* y la creación de DAOs para lo que hizo falta la tarea siguiente.
- **Estudio de *Solidity* y *Javascript*:** Fue necesario estudiar estos lenguajes de programación y su interacción con *Ethereum* para entender el funcionamiento del proyecto.
- **Planteamiento final del proyecto:** Debido a la complejidad que conlleva la creación de una herramienta como la propuesta en primera instancia, se acordó utilizar los conocimientos adquiridos para la mejora de un sistema que implementase todo lo estudiado, incluyendo el uso de IPFS.
- **Estudio de IPFS:** La mejora propuesta requería del estudio de este nuevo sistema y de la interacción con el mismo. Además se decidió utilizar un cifrado de la información para la subida de los datos eligiendo *openpgp.js* por sus características.
- **Adaptación e inserción de IPFS en el código:** Este periodo fue dedicado al uso y elección de las APIs de IPFS para poder comunicarse con el nodo a través de *javascript* y a la adaptación del protocolo de encriptación en el código del proyecto para cumplir con los objetivos propuestos.
- **Pruebas:** Se realizaron todas las comprobaciones necesarias para demostrar el correcto funcionamiento del sistema completo.
- **Redacción de la memoria:** Esta fase tuvo varias etapas, ya que se comenzó escribiendo los conceptos teóricos estudiados para la implementación y; posteriormente, se decidió realizar la memoria en *Latex* con el fin de mejorar la visualización y el formato del trabajo.

Todos los pasos seguidos han sido supervisados por el tutor mediante reuniones periódicas para comprobar si se cumplían los objetivos propuestos y adaptar el trabajo al nivel y al tiempo del que se disponía. Debido a ello se decidió aprovechar los conocimientos teóricos adquiridos durante la primera

etapa del trabajo y llevarlos a la práctica insertando nuevas funcionalidades en un proyecto de *blockchain* real.

2.1. Costes

En este apartado se pretende dar un coste aproximado del proyecto en función de las horas invertidas y los requisitos del trabajo. El sistema se ha basado en herramientas de código abierto, lo que implica que los costes por *software* serían mínimos. Aún así hay que incluir: el precio del equipo donde se ha implementado, la licencia de *windows* 10, el mantenimiento, el consumo eléctrico, la oficina con acceso a *internet*, posibles gastos extras y el sueldo de los ingenieros.

Todos estos aspectos se recogen en la Tablas 2.1, 2.2, 2.3 pudiendo recurrir a la Tabla 2.4 para ver el coste total del proyecto.

Sueldo ingenieros	Horas	Coste a la hora	Coste total
Junior	480	30€/h	14400€
Superior	20	70€/h	14000€
Subtotal			28400€

Tabla 2.1: Mano de obra.

Requisitos	Unidades	Coste total
Equipo	1	1100€
Licencia Windows 10	1	145€
Licencia Ganache	1	0€
Licencia Truffle	1	0€
Licencia IPFS	1	0€
Subtotal		1245€

Tabla 2.2: Material.

Concepto	Meses	Coste al mes	Coste total
Internet	3	50€/mes	150€
Electricidad	3	80€/mes	240€
Mantenimiento	3	60€/mes	180€
Alquiler oficina	3	600€/mes	1800€
Gastos extras			250€
Subtotal			2620€

Tabla 2.3: Costes indirectos.

Tipo de coste	Coste total
Mano de obra	28400€
Material	1245€
Costes indirectos	2620€
TOTAL	32265€

Tabla 2.4: Coste total del proyecto.

Capítulo 3

Análisis del problema

3.1. Especificación de requisitos

El proyecto consta de dos funcionalidades muy bien definidas:

- Poder almacenar una evidencia de un delito informático en una *blockchain*.
- Poder recuperar a través de un enlace IPFS la información original en caso de modificación o pérdida de los datos denunciados.

Para conseguir el primer punto, se ha partido de otro trabajo de fin de grado alojado en <https://github.com/rubcv/TFG>. En dicho proyecto se propone un sistema de almacenamiento basado en *Blockchain* donde se guarda un *hash* del contenido HTML de la *web* (donde se haya cometido el delito) con el fin de comprobar si las pruebas recogidas en primera instancia y las presentadas en el juicio posterior han cambiado.

En lo que respecta al segundo punto y en base al proyecto anterior, se pretende solucionar el problema de que la información presentada en el juicio haya sido manipulada y por tanto el *hash* no coincida. También se soluciona el posible robo de evidencias ya que se facilita un enlace a una base de datos inmutable y descentralizada que contiene las pruebas denunciadas.

Es muy importante aclarar que este trabajo propone un sistema destinado a la denuncia de delitos informáticos realizados a través de **páginas webs**.

Los pasos que se siguen son:

1. El **usuario** accede a una **interfaz web** donde introduce la **URL** de la página donde se encuentra el contenido a denunciar.
2. La interfaz web recurre al **servidor** que envía la URL al **Cliente**.

3. El Cliente mediante el *smart contract* se pondrá en contacto con el **Oráculo** que descargará el contenido **HTML** de la URL introducida por el usuario.
4. El Oráculo cifrará el contenido devuelto con el protocolo de encriptación **SHA-256** y además, subirá a **IPFS**: el contenido HTML cifrado con una clave pública y la clave privada cifrada para poder descryptar la información.
5. El Oráculo envía al Cliente:
 - El *hash* del contenido HTML que servirá para evidenciar que la información subida a IPFS es la original.
 - El enlace de IPFS donde se aloja la **información cifrada** con la clave pública.
 - El otro enlace de IPFS que contiene la **clave privada cifrada**.

Además del título de la *web* y la fecha en formato UNIX, quedando todos estos datos almacenados en la *blockchain*.

Es imprescindible el archivo **COMPROBACIÓN** ya que es donde se descifra el contenido de IPFS y donde se realiza de nuevo el *hash* con el protocolo SHA-256 para comprobar que la información de IPFS es la original. De manera que se puede consultar la información denunciada asegurando que no se ha modificado.

El usuario necesitará un monedero de *Ethereum* ya que el *smart contract* requiere de un gasto de *gas* como se comenta en 1.4.3.

La comunicación entre el Cliente, el *smart contract* y el Oráculo se realiza mediante una API de *Javascript* denominada **Web3** que incluye las funcionalidades necesarias para la interacción con la EVM de *Ethereum*.

3.2. Análisis

El motivo por el cual no se ha almacenado directamente el contenido de la página *web* en la *blockchain* se debe a la limitación de almacenamiento de los bloques impuesta por el sistema para posibilitar la sincronía en la minería. Así pues, la opción elegida para almacenar la información de forma segura ha sido IPFS.

El hecho de ser un sistema descentralizado, hace que en IPFS sea imposible saber si la información ha sido eliminada de forma permanente ya que cabe la posibilidad de que esté almacenada en cualquier nodo o cualquier usuario puede haberla descargado. A donde se quiere llegar es que es una base de datos pública donde cualquiera puede obtener el contenido almacenado si conoce el enlace del archivo. Ante esta característica surge

el problema de que si se pretende subir información de cuentas privadas se podría estar infringiendo la ley de protección de datos de los usuarios. Así pues se ha decidido subir la información cifrada con una clave pública. En este punto se ha presentado el inconveniente de tener que almacenar la clave privada en algún sitio seguro ya que es bastante larga y difícil de recordar. Con lo cual también se ha almacenado en IPFS pero cifrada con una sola palabra que sea fácil de memorizar por parte del usuario.

A la hora de elegir un cliente de IPFS se ha optado por el que está programado en **Go** (lenguaje de programación inspirado en *C* y desarrollado por *Google*) ya que el que está programado en **Javascript** presenta el inconveniente de tener que descargar los archivos de IPFS para obtener su contenido.

Para acabar con este apartado se ha de especificar que el almacenamiento de las evidencias y de los enlaces a IPFS se ha hecho en una *blockchain* de prueba local como lo es Ganache 1.4.5 ya que no se dispone de los recursos para montar una *blockchain* privada y respecto a las páginas "denunciadas" han sido de carácter público excepto las pertenecientes a redes sociales que se han utilizado cuentas de prueba creadas para este trabajo.

Capítulo 4

Diseño

Antes de comenzar se debe aclarar que la estructura de la *blockchain* junto con el servidor y la interfaz *web* utilizados pertenecen al trabajo de fin de grado mencionado en 3. Así pues, en este proyecto se ha insertado la comunicación con la red IPFS aplicando los cambios necesarios para su funcionamiento. Por lo tanto en este capítulo se exponen los conceptos clave del anterior proyecto junto con las incorporaciones requeridas para cumplir con los objetivos propuestos.

4.1. Diseño global del sistema

Para facilitar la comprensión se en esta sección se explican las especificaciones de cada apartado ilustrado en la Figura 4.1.

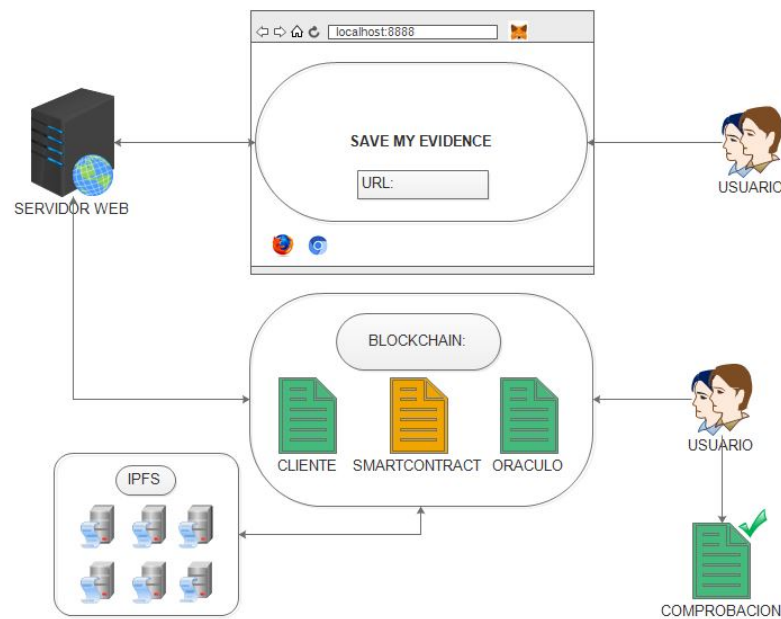


Figura 4.1: Esquema general del proyecto

El primer paso consiste en la comunicación del usuario con la **interfaz web** donde se debe introducir la URL de la página cuyo contenido se quiere denunciar.

Dicha interfaz pertenece al **servidor web** desarrollado con el *framework* **Express** de **Node**. Para más información se puede recurrir a <https://www.npmjs.com/package/express>.

La principal función de esta interfaz además de facilitar la URL al servidor que interactuará con la *blockchain*, es comprobar si el navegador donde se aloja el servidor tiene instalado **Metamask**. Como se comenta en 1.4.7, para poder habilitar dicha extensión es imprescindible que el navegador utilizado sea *Firefox* o alguno basado en *Chromium*. De esta manera el procedimiento seguido sería el que se muestra en la Figura 4.2

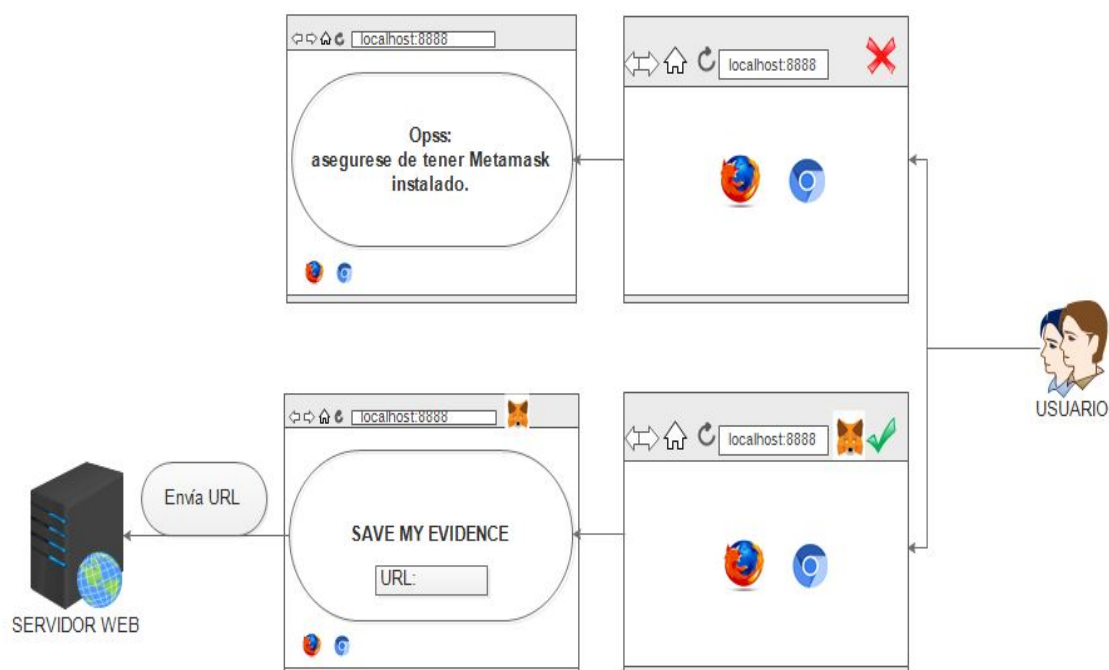


Figura 4.2: Comportamiento de la interfaz gráfica.

Una vez que el servidor obtiene la URL, comprueba si el formato es correcto, espera a la confirmación del usuario en Metamask y envía la URL a la *blockchain* para que se pueda crear la evidencia deseada. Una vez hecho esto, notifica al usuario que la evidencia ha sido almacenada.

En este momento es cuando empieza la interacción entre los ficheros dentro de la *blockchain* escogida: **Ganache**.

Como se puede ver en la Figura 4.3, cuando el servidor envía la URL al **Cliente** y al **Oráculo**, es el *Cliente* el que prosigue solicitando el almacenamiento de la evidencia al *Oráculo* por medio del *smart contract*. Estas comunicaciones se realizan mediante **eventos** como se explicará en el Capítulo 5. Seguidamente el *smart contract* le envía la petición al *Oráculo*.

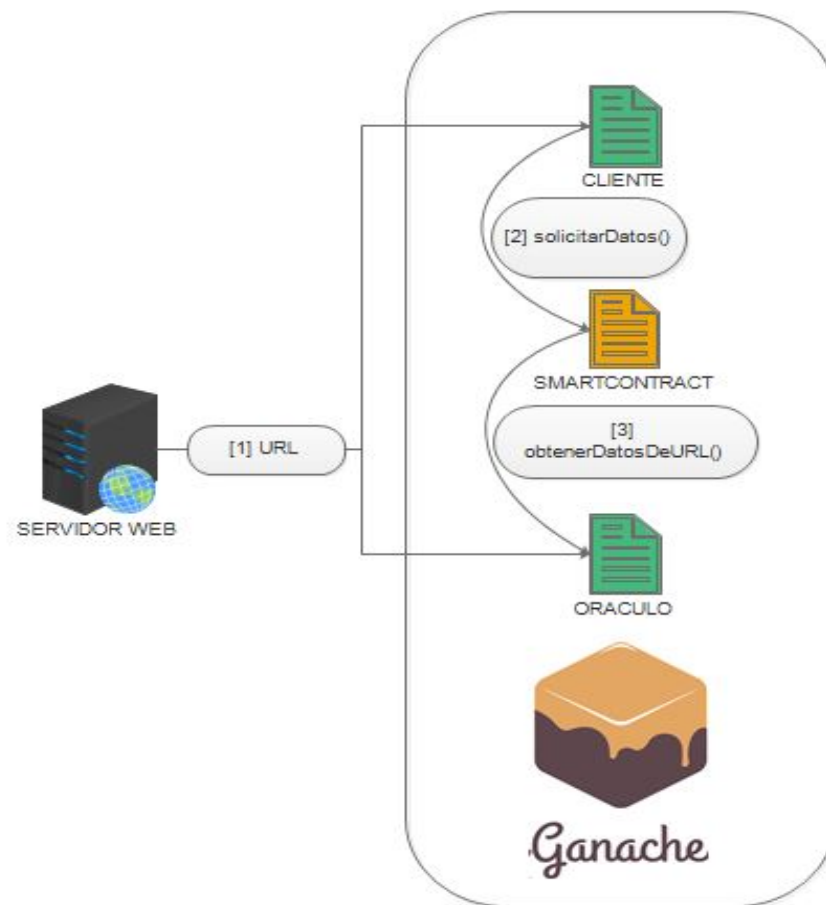


Figura 4.3: Primeras interacciones en Ganache

A continuación es el Oráculo el que se encarga de solicitar el contenido de la URL de fuera de la *blockchain*, hacerle el *hash*, cifrar el HTML y subirlo a IPFS junto con la clave para descifrarlo como puede verse en la Figura 4.4.

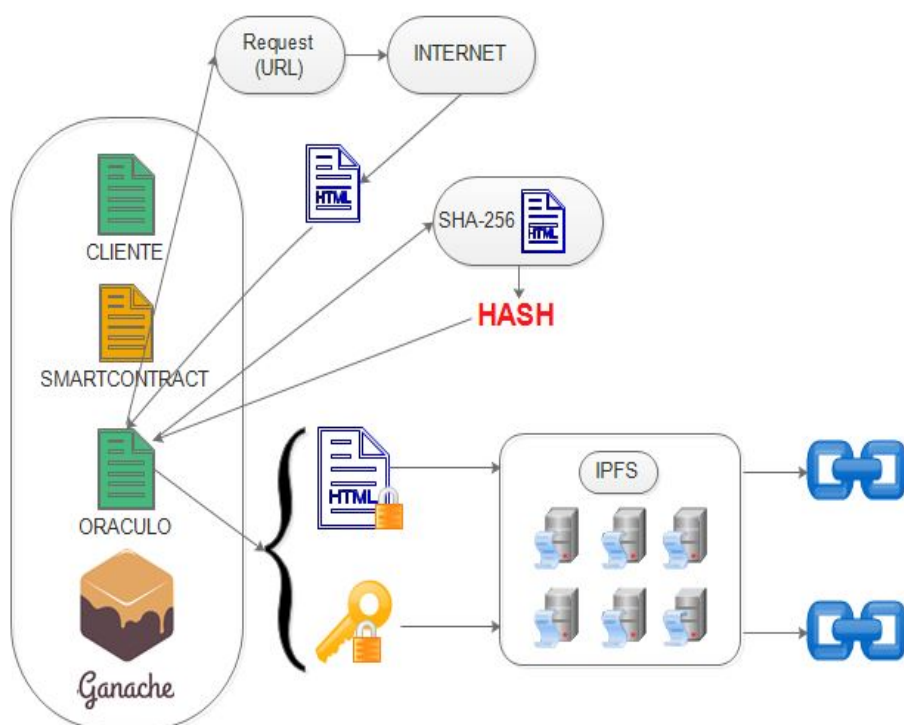


Figura 4.4: Funciones del Oráculo

Posteriormente el Oráculo envía al *smart contract* mediante la función *guardarSolicitud()*:

- El **título** de la página Web la cual se quiere denunciar su contenido.
- El **hash** resultante para posteriormente poder comprobar si la información ha cambiado.
- La **hora** de la operación en formato *UNIX*
- El **enlace** de IPFS donde se encuentra el HTML encriptado.
- El **enlace** de IPFS donde se encuentra la clave privada cifrada necesaria para descryptar la información.

Por lo tanto a través del *smart contract* la información se queda almacenada en **Ganache**. Por último, el *smart contract* notifica al Cliente que la **evidencia ha sido almacenada** y este lo envía al servidor para que se lo haga llegar al usuario a través de la interfaz web. Todo este proceso se muestra en la Figura 4.5

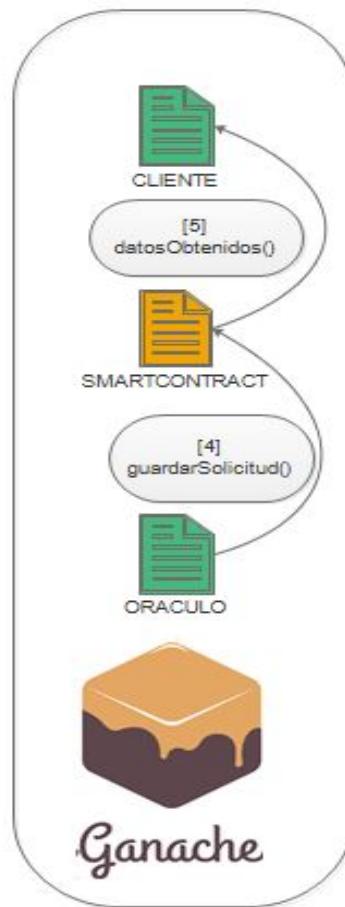


Figura 4.5: Últimas interacciones en Ganache.

Una vez que se ha almacenado la evidencia en la *blockchain* se necesita un *script* que se encargue de descargar la información de IPFS, la descifre y le haga el *hash* para asegurar la veracidad de los datos en el día que se deban presentar las pruebas. Para ello se ha creado el fichero **COMPROBACIÓN** donde el usuario solo tiene que introducir los dos **enlaces** a IPFS que se muestran en la función *guardarSolicitud()* y el *string* utilizado para cifrar la clave privada (conocido únicamente por el usuario) . En la Figura 4.6 se representan los pasos que sigue este fichero, obteniendo finalmente el *hash* el cual se espera que sea igual al almacenado en Ganache.

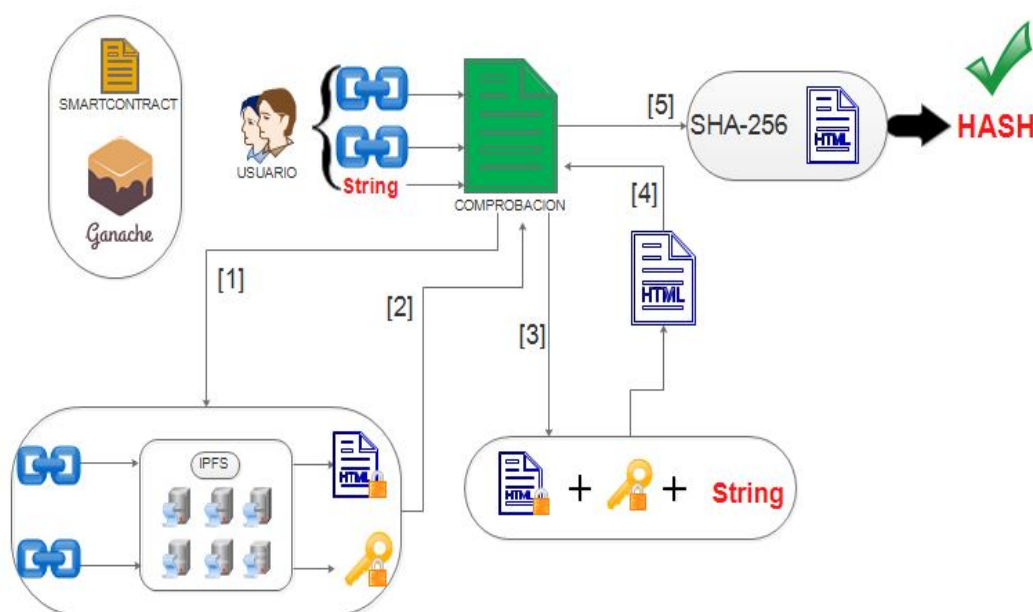


Figura 4.6: Función del fichero COMPROBACIÓN.

4.2. Especificaciones IPFS

Para entender mejor la interacción con IPFS a continuación se explican los requisitos para su funcionamiento. Se ha de puntualizar que al ser una tecnología en constante desarrollo, en este trabajo se ha elegido el cliente de IPFS programado en el lenguaje *Go*. Aunque casi la totalidad del sistema está escrita en *Javascript*, se ha elegido el anterior porque es el único que tiene todas las funcionalidades completas del sistema. Además, a la hora de solicitar los datos de IPFS en el fichero *COMPROBACION*, este devuelve la información sin necesidad de confirmar la descarga, como lo hace el nodo escrito en *Javascript*.

La *API* utilizada para la comunicación con el nodo y el *Oráculo* a través de *http*, sí está escrita en *Javascript* ya que es la única desarrollada al completo como se explicará al final de esta sección.

Para comenzar, una vez descargado el cliente de IPFS programado en *Go*, se debe inicializar el nodo. Esto se realiza con el comando visto en el Listado de código 4.1 a través del terminal en modo **administrador**.

```
1 $: ipfs init
```

Listado de código 4.1: Inicializar el perfil de IPFS

Una vez ejecutado el comando, en el terminal aparece lo ilustrado en la Figura 4.7.

```
initializing IPFS node at C:\SPB_Data\.ipfs
generating 2048-bit RSA keypair...done
peer identity: QmQGYFySKtruH9H5zLe7Dr5s5rcBXyYc9zqTGKzMKM6HsF
to get started, enter:

    ipfs cat /ipfs/QmS4ustL54uo8FzR9455qaxZwuMiUhyvMcX9Ba8nUH4uVv/readme
```

Figura 4.7: Inicio IPFS

A continuación se debe introducir el comando recomendado por por IPFS. En este caso sería el mostrado en la Línea de código 4.2.

```
1 $: ipfs cat /ipfs/QmS4ustL54uo8FzR9455qaxZwuMiUhyvMcX9Ba8nUH4uVv/readme
```

Listado de código 4.2: Mostrar página de bienvenida de IPFS

Se abrirá una ventana de bienvenida vista en la Figura 4.8a partir de la cuál se podrá comenzar a interactuar con el nodo.

```
Hello and Welcome to IPFS!

IPFS

If you're seeing this, you have successfully installed
IPFS and are now interfacing with the ipfs merkledag!

-----
Warning:
  This is alpha software. Use at your own discretion!
  Much is missing or lacking polish. There are bugs.
  Not yet secure. Read the security notes for more.
-----

Check out some of the other files in this directory:

./about
./help
./quick-start    <-- usage examples
./readme         <-- this file
./security-notes
```

Figura 4.8: Ventana de bienvenida de IPFS

Se debe inicializar el *daemon* para tener acceso a *internet* e interactuar con otros nodos, por lo que se debe ejecutar el comando visto en Listado de código 4.3.

```
1 $: ipfs daemon
```

Listado de código 4.3: Inicializar el daemon

Observando la información devuelta por IPFS en la Figura 4.9 se debe recalcar la dirección por defecto de la **API** y el **gateway** ya que se utilizará en el siguiente capítulo.

```
Initializing daemon...
go-ipfs version: 0.4.22-
Repo version: 7
System version: amd64/windows
Golang version: go1.12.7
Swarm listening on /ip4/127.0.0.1/tcp/4001
Swarm listening on /ip4/169.254.151.89/tcp/4001
Swarm listening on /ip4/169.254.164.162/tcp/4001
Swarm listening on /ip4/169.254.69.66/tcp/4001
Swarm listening on /ip4/172.20.122.152/tcp/4001
Swarm listening on /ip6:::1/tcp/4001
Swarm listening on /p2p-circuit
Swarm announcing /ip4/127.0.0.1/tcp/4001
Swarm announcing /ip4/169.254.151.89/tcp/4001
Swarm announcing /ip4/169.254.164.162/tcp/4001
Swarm announcing /ip4/169.254.69.66/tcp/4001
Swarm announcing /ip4/172.20.122.152/tcp/4001
Swarm announcing /ip6:::1/tcp/4001
API server listening on /ip4/127.0.0.1/tcp/5001
WebUI: http://127.0.0.1:5001/webui
Gateway (readonly) server listening on /ip4/127.0.0.1/tcp/8080
Daemon is ready
```

Figura 4.9: Inicialización del daemon

Si se desea acceder a la interfaz *web* de IPFS se debe introducir en el navegador la dirección de la **WebUI**: `http://127.0.0.1:5001/webui` donde se devolverá un contenido parecido al de la Figura 4.10

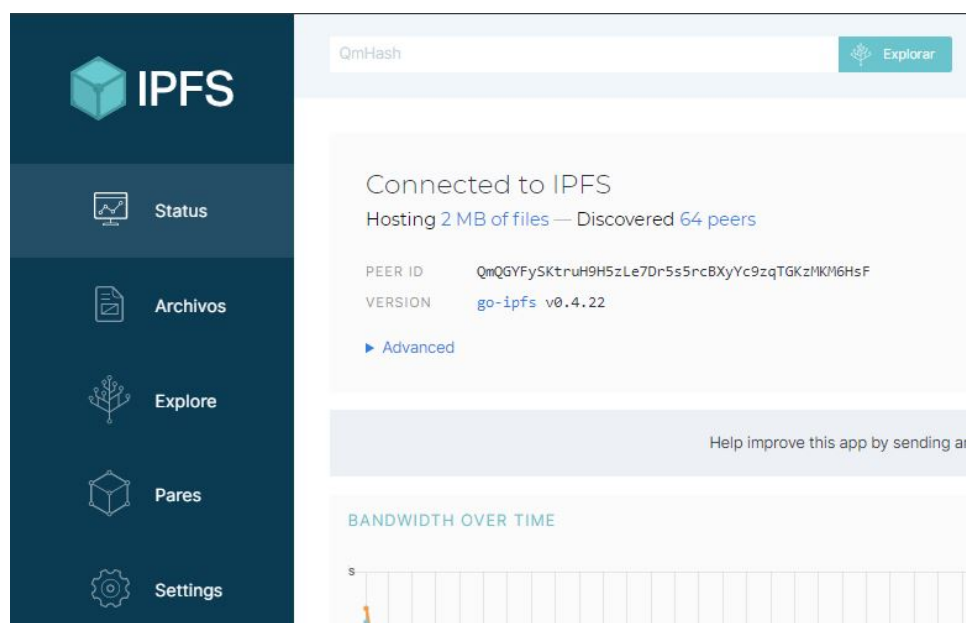


Figura 4.10: Interfaz web de IPFS

Por último, para la insercción de IPFS en este proyecto se ha reccurido a la **API** *ipfs-http-client* escrita en *javascript*. Se ha elegido esta API ya que el código del proyecto está escrito en este lenguaje y por suerte es de las pocas APIs completamente desarrolladas actualmente: Noviembre de 2019. Para más información se puede visitar <https://github.com/ipfs/ipfs#project-links>.

Capítulo 5

Implementación

En este capítulo se explica en detalle el código del proyecto a fin de entender el funcionamiento expuesto en el capítulo anterior.

5.1. Javascript

Javascript se caracteriza por el uso de funciones asíncronas idóneas para comunicaciones cliente-servidor llamadas ***callbacks***. Estas funciones asíncronas se ejecutan en hilos paralelos al hilo principal del código donde se encuentran, lo que permite realizar varias funcionalidades a la vez. El problema llega cuando se necesita una estructura secuencial en el código con funciones asíncronas ya que si se requiere un dato cuyo valor tarda tiempo en devolverse, el código no espera a obtener dicho valor ya que realiza esta ejecución en segundo plano.

Como solución a este inconveniente, se necesita anidar unos callbacks con otros creando lo que se llama el *callbacks hell*. Un ejemplo de código secuencial asíncrono mediante funciones *callbacks* puede verse en el Listado de código 5.1.

```
1 function printAll(){
2   printString("A", (error, res) =>{
3     if(error){
4       console.log(error)
5     }else{
6       printString("B", (error, res) =>{
7         if(error){
8           console.log(error)
9         }else{
10          printString("C", (error, res) =>{
11            ...
12          })
13        }
14      })
15    })
16  }
17 }
18 })
```

```
19 }  
20  
21 printAll()
```

Listado de código 5.1: Ejemplo callback hell

Para realizar esta estructura secuencia evitando escribir el código anterior ya que puede resultar incómodo e induce a error, *javascript* creó lo que se conoce como **promise**. Una *promise* es un tipo de función que permite realizar llamadas asíncronas a través de la nomenclatura mostrada en el Listado de código 5.2. En este tipo de funciones, el comando *then()* se ejecuta cuando la promesa anterior ha sido resuelta satisfactoriamente, y el comando *catch()* cuando se produce un error.

```
1 function printAll() {  
2   printString("A")  
3   .then(() => printString("B"))  
4   .then(() => printString("C"))  
5   .catch((error) => {console.log(error)})  
6 }  
7 printAll()
```

Listado de código 5.2: Ejemplo Promise

Por último, *javascript* introdujo un librería llamada **async/await** que permite realizar lo anterior de manera más simple como puede verse en el Listado de código 5.3

```
1 async function printAll() {  
2   try{  
3     await printString("A")  
4     await printString("B")  
5     await printString("C")  
6   } catch(error) {console.log(error)}  
7 }  
8 printAll()
```

Listado de código 5.3: Ejemplo async/await

A lo largo del código del proyecto se verán estas tres nomenclaturas para realizar distintas acciones. Se ha de aclarar que las funciones creadas en los ejemplos anteriores se ejecutarán con dicha estructura secuencial pero, si existen otras funciones dentro del código, estas últimas no esperarán a que acaben las de los ejemplos ya que se realizan en segundo plano.

5.2. Código del almacenamiento de la evidencia

En esta sección se explican las funciones más importantes para conseguir la implementación del diseño. Como el proyecto está basado en otro trabajo de fin de grado donde se implementa el sistema completo pero sin la incorporación de IPFS y del cifrado de la información, se habla de las funciones principales para su funcionamiento profundizando sobre todo en las correspondientes a estas innovaciones. Si se quiere saber más so-

bre aspectos no comentados en esta sección se puede consultar en <https://github.com/rubcv/TFG>.

Para comenzar, una vez que el usuario introduce la **URL** en la interfaz *web* y el servidor comprueba su validez, este último le pasa el enlace al *Cliente* y al *Oráculo* mediante el código mostrado en el Listado de código 5.4 perteneciente al fichero *index.js*.

```
1 startOracle.startOracle(urlFormulario);
2 startClient.startClient(urlFormulario);
```

Listado de código 5.4: Comienzo del Cliente y el Oráculo

Lo primero que hacen tanto el Cliente como el Oráculo es subscribirse a los **eventos** *datosObtenidos()* y *obtenerDatosDeURL()* respectivamente. Estos eventos los emite el *smart contract* cuando el *Cliente* y el *Oráculo* recurren a él mediante las funciones *solicitarDatos()* y *guardarSolicitud()*.

Por tanto el *Cliente* mediante la función *solicitarDatos()* envía al *smart contract* la URL y la fecha de ejecución a través de la función vista en el Listado de código 5.5. Esta función se ejecuta en el fichero *Cliente.js*

```
1 ethereum.contract.solicitarDatos(urlFormulario, fecha_unix);
```

Listado de código 5.5: Función mediante la cual el Cliente solicita los datos al Smartcontract

Seguidamente el *smart contract* cuyo fichero corresponde a *Contract.sol*, emite el evento *obtenerDatosDeURL()* el cual es escuchado por el *Oráculo* con las variables de entrada: URL y fecha. Esta función puede verse en el Listado de código 5.6.

```
1 function solicitarDatos(string memory URL_solicitada, uint256 currentDate)
2 public payable {
3     emit obtenerDatosDeURL(URL_solicitada, currentDate);
4 }
```

Listado de código 5.6: Evento que escucha el Oráculo

Es a partir de este momento en donde se ha modificado el contenido del proyecto comentado al principio de la sección. El *Oráculo* es el encargado de realizar las funciones descritas en el capítulo 4 por lo que al haber incorporado nuevas funcionalidades, se ha requerido la alteración del código. Es muy importante explicar que el *Oráculo* empieza a ejecutarse una vez que el servidor le envía la URL, pero no realiza ninguna acción hasta que el *smart contract* emite el evento.

Continuando con la emisión de *obtenerDatosDeURL()*, el fichero *Oracle.js* realiza la función genérica *obtenerDatosCallback()* que contiene las funciones mostradas en el pseudocódigo 5.7.

```
1 await obtenerDatosDeURL(function (res, error) { L
2     if (error) {
3         console.log(error);
4     } else {
5         obtenerDatosCallback(options);
```

```

6   }
7   });
8
9   async function obtenerDatosCallback(options){
10     const html = await request(options);
11     const datosJSON = await obtenerJSON(html);
12     await guardarSolicitud(datosJSON)
13     .catch(error);
14   };

```

Listado de código 5.7: Activación del evento y función general del Oráculo

Como puede observarse utiliza la metodología descrita en 5.1 mediante las funciones *async/await*. Esto quiere decir que esperarán la finalización de la tarea anterior de forma secuencial. En este punto es importante recordar que debido a que en el fichero *Oracle.js* no se realiza ninguna otra función que la genérica: *obtenerDatosCallback()*, no hay que preocuparse sobre la finalización paralela de otras funciones.

La variable de entrada de la función *obtenerDatosCallback()* es la **URL** facilitada por el servidor, aunque como se ha dicho anteriormente, el código no se ejecutará hasta que el *smart contract* emita el evento *obtenerDatosDeURL()*. Fijándose en en Listado de código 5.7, la primera función que realiza es un *request* a la URL de entrada para conseguir el **HTML** de la *web*. Una vez obtenido el HTML en la variable *html*, esta se pasa a la función *obtenerJSON()*. He aquí la función que conlleva más tiempo de ejecución ya que como se verá a continuación, es donde se realiza el proceso de *hasheo*, el cifrado y la subida de los datos a IPFS.

Para facilitar el seguimiento de dicha función, en el Listado de código 5.8 se puede ver el proceso que sigue.

```

1   async function obtenerJSON(data){
2     const numeroBits = 512
3     const nombre = 'salva'
4     const palabra_aleatoria = 'TFG'
5     const [cifrado, claveprivada] = await cifrar(data, numeroBits, nombre,
6       palabra_aleatoria)
7     var datosJSON = {
8       "WEB" : await obtenerWeb(data),
9       "HASH" : await sha256(data),
10      "DATOS" : await linkipfs(cifrado),
11      "CLAVEPRIVADA" : await linkipfs(claveprivada),
12    }
13    console.log(datosJSON)
14    return datosJSON
15  };

```

Listado de código 5.8: Función del Oráculo que conlleva más tiempo de ejecución

Primero se declaran las variables necesarias para el cifrado:

- *numeroBits*: Determina el tamaño de la clave utilizada. Se debe dar el valor de un número que sea potencia de dos y mayor que 2^9 .

- *nombre*: Nombre del usuario que cifra la información.
- *palabra aleatoria*: Es la palabra con la que se cifrará la clave privada que se sube a IPFS y que solo la deberá conocer el usuario que ejecute el sistema.

Una vez declaradas las variables de entrada de la función *cifrar()*, esta devuelve:

- El **HTML cifrado** con la clave pública.
- La **clave privada** (necesaria para el descifrado del HTML) **cifrada** con la palabra aleatoria introducida.

A continuación se crea la lista de datos que se mostrarán en *Ganache* en formato **JSON**:

- *"WEB"*: Se parsea el HTML de la web y se devuelve el título.
- *"HASH"*: Se realiza el *hash* del HTML con el protocolo SHA-256, de manera que se almacene la evidencia de la información original.
- *"DATOS"*: Se devuelve el enlace de IPFS donde se encuentra el HTML cifrado.
- *"CLAVEPRIVADA"*: Se devuelve el enlace de IPFS donde se encuentra la clave privada cifrada.

Para la devolución de los enlaces a IPFS se ha creado la función *linkipfs()*. Esta función consta de varias partes para su funcionamiento como puede verse en el pseudocódigo 5.9

```
1 const IPFS = require('ipfs-http-client');
2
3 var node = new IPFS('localhost', '5001', { protocol: 'http' });
4
5 async function linkipfs (cifrado) {
6   files = await node.add(IPFS.Buffer.from(cifrado))
7   var link = "https://ipfs.io/ipfs/" + files[0].hash;
8
9   return link
10 }
```

Listado de código 5.9: Subida de datos y devolución del enlace IPFS

Primero se crea el objeto **"IPFS"** que contiene la librería de la API de *javascript*: **"ipfs-http-client"**. Para su utilización se necesita que el *daemon* explicado en 4.2 esté activado. Seguidamente se crea la instancia **"node"** donde se carga la dirección de la API, que en este caso es la que viene inicialmente en el archivo de configuración. Por último se suben los datos contenidos en la variable de entrada y se devuelve el enlace de IPFS.

A este enlace se ha añadido el *gateway público* "ipfs.io" por defecto, pero se podría acceder al fichero desde cualquier nodo de IPFS.

Siguiendo con la ejecución, una vez que se tiene la variable *datosJSON* con la información que se mostrará en la *blockchain*, continua la función *obtenerDatosCallback()* del Listado de código 5.7 con "guardarSolicitud()". Esta es la última tarea que ejecuta el *Oráculo* con la que manda los datos al *smart contract* para que quede almacenado en la *blockchain*. Como puede verse en el pseudocódigo 5.10 a parte de la información contenida en *datosJSON*, envía la fecha en formato UNIX y la versión del sistema.

```

1 async function guardarSolicitud(datosJSON) {
2
3   let fecha = new Date().getTime();
4   let fecha_unix = fecha / 1000;
5
6   var datos = JSON.stringify(datosJSON);
7
8   var version = 1;
9
10  ethereum.contract.guardarSolicitud(fecha_unix, datos, version)
11
12 };

```

Listado de código 5.10: Función mediante la cual el Oráculo envía los datos al Smartcontract

Para acabar con lo referente a la *blockchain*, una vez que el *smart contract* recibe los datos, comprueba que provienen de la dirección del *Oráculo* y emite el evento *datosObtenidos()* que escucha el *Cliente*. Esto se muestra en el Listado de código 5.11.

```

1 function guardarSolicitud(uint256 timestamp, string memory datosJSON,
2   uint128 version) public {
3   require(msg.sender == oracleAddress,
4     "Solicitud no realizada por el Oraculo"
5   );
6   emit datosObtenidos(timestamp, datosJSON, version);
7 }

```

Listado de código 5.11: Evento que escucha el Cliente

5.3. Código de comprobación y descarga de la evidencia

En esta sección se explica el código del fichero **COMPROBACIÓN.js**. Este fichero se ha creado para posibilitar el acceso de la información el día del juicio y comprobar que corresponde con la evidencia almacenada. Lo primero que ha de modificar el usuario son los valores de las variables de entrada mostradas en el Listado de código 5.12

```

1 const DATOS = 'http://ipfs.io/ipfs/
   Qme2W8ft4faMgkFvcRzMHbj9J6WtUyyTcQq4bFSqqUt2dc'

```

```
2
3 const CLAVEPRIVADA = 'http://ipfs.io/ipfs/
  QmX4QgS1DXGFjS7jctQSEvJXWUb5Wk3FYoAA6TiG1F5ysD'
4
5 const STRING = 'TFG'
6
7 main(DATOS, CLAVEPRIVADA, STRING)
```

Listado de código 5.12: Introducción de los datos en el fichero COMPROBACIÓN.js

Notar que la variable "*STRING*" contiene la **palabra aleatoria** necesaria para descifrar la clave privada.

Seguidamente se muestra en el Listado de código 5.13 el contenido de la función *main()* con las variables de entrada ya establecidas.

```
1 async function main (urldata, urlkey, palabra_aleatoria){
2   const mensaje_cifrado = await request(urldata)
3   const privada = await request(urlkey)
4   const privKeyObj = (await openpgp.key.readArmored(privada)).keys[0]
5
6   await privKeyObj.decrypt(palabra_aleatoria)
7
8   const descifrar ={
9     message: await openpgp.message.readArmored(mensaje_cifrado),
10    privateKeys: [privKeyObj]
11  }
12
13  const descifrado = await desencriptar(descifrar)
14  const hash = await sha256(descifrado.data)
15
16  console.log(descifrado)
17  console.log(hash)
18
19 }
```

Listado de código 5.13: Contenido función main()

Se observa que las dos primeras líneas de código dentro de la función están dedicadas a la descarga de la información contenida en los enlaces de IPFS. Es importante denotar que el tiempo de respuesta del código puede ser largo ya que la red IPFS debe encontrar todos los paquetes utilizados para el almacenamiento de los datos.

A continuación se descifra la clave privada con la palabra aleatoria escogida por el usuario en el momento en el que se almacenó la evidencia en la *blockchain*.

Una vez descifrada la clave privada, se crea el constructor ***descifrar*** que contiene el HTML cifrado y dicha clave privada y se envía como variable de entrada de la función *desencriptar()* que devuelve el **HTML original**.

Por último, para comprobar que la información descargada corresponde con los datos denunciados y almacenados en *Ganache*, se realiza un *hash* del HTML con el protocolo SHA-256 que debe coincidir con el mostrado en la *blockchain*.

Capítulo 6

Evaluación y pruebas

En este capítulo se realiza a modo de ejemplo una supuesta denuncia de un mensaje privado a una cuenta de *Twitter* para comprobar el funcionamiento del sistema. Como se dijo en 3.2, las cuentas utilizadas han sido creadas en modo de prueba para este proyecto,

El nombre de las cuentas utilizadas han sido:

- @Prueba1Tfg
- @Prueba2T

La primera cuenta le envía un mensaje privado a la segunda que se supone ofensivo aunque el mensaje sea: *Esto es una prueba* como puede verse en la Figura 6.1.

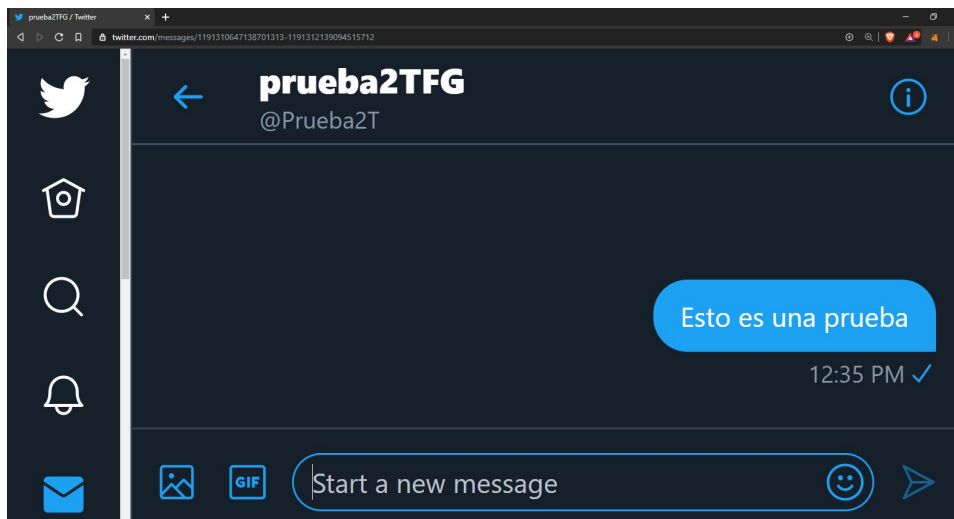


Figura 6.1: Mensaje enviado de @Prueba1Tfg a @Prueba2T

El usuario de la segunda cuenta procede a realizar la denuncia mediante el sistema expuesto a lo largo de la memoria (recordar que según el contexto del proyecto, esto debería hacerlo un agente).

Siguiendo el procedimiento descrito en A, lo primero sería acceder al enlace cuya página contiene el mensaje a denunciar como se muestra en la Figura 6.2.

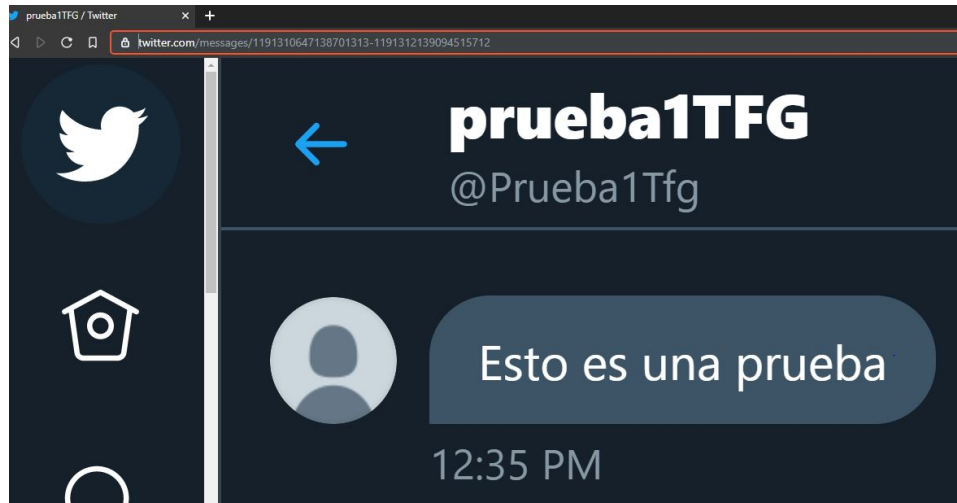


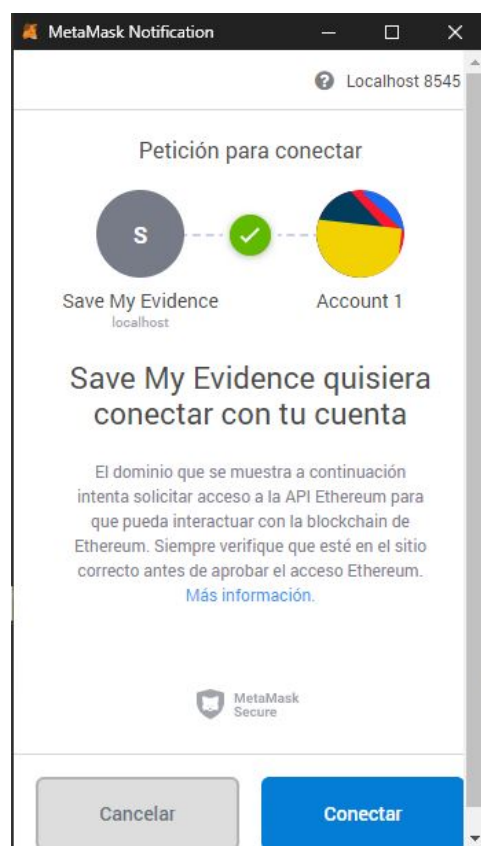
Figura 6.2: Página cuyo contenido se quiere denunciar

A continuación se debe introducir dicho enlace en la interfaz *web* ilustrado en la Figura 6.3.



Figura 6.3: Funcionamiento de la interfaz *web*

Seguidamente se debe confirmar el pago realizado en la ventana emergente de *Metamask* como se ve en la Figura 6.4.

Figura 6.4: Confirmación del pago en *Metamask*

Si se recurre a la interfaz *web* puede observarse la confirmación de que el proceso se ha ejecutado con éxito mostrado en la Figura 6.5.

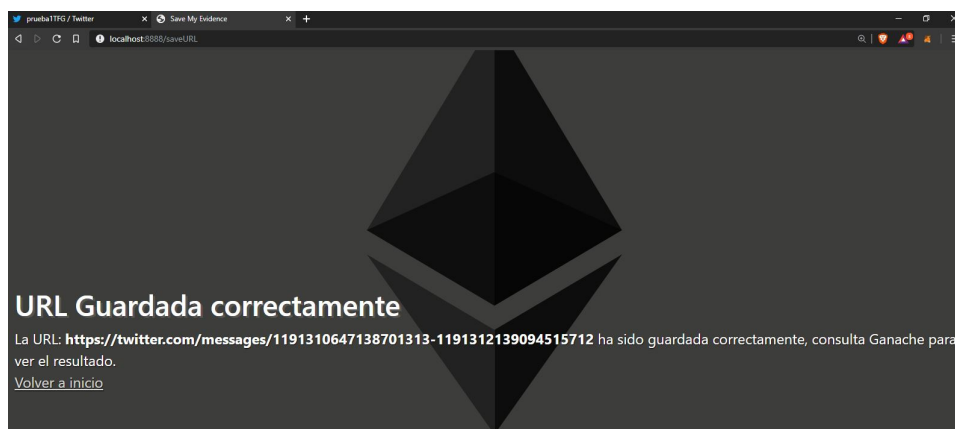


Figura 6.5: Notificación de éxito del almacenamiento

Por último, se debe acceder a *Ganache* para ver el almacenamiento de la evidencia mostrado en la Figura 6.6 .

CONTRACT	ADDRESS
Contract	0xD67D7F7465aBB2A610Bc926c8BaBf930903622d6
FUNCTION	
guardarSolicitud(timestamp: uint256, datosJSON: string, version: uint128)	
INPUTS	
1572868834, {"WEB":"Iniciar sesión en Twitter","HASH":"c2ac884de2eabe95ab139f1257ba3171c037c1c9a5984a95ff8c1c1d6c819e43","DATOS":"https://ipfs.io/ipfs/QmdLRBtxHNm2CirRdczEc4vSrN5XrGjNWvLbwcTNfkPTUf","CLAVEPRIVADA":"https://ipfs.io/ipfs/QmQhk3wXrxdXrT8jKecV8y7x78KFnpio313jYpNNmoeg"}, 1	

Figura 6.6: Bloque de *Ganache* donde se almacena la evidencia

De la información devuelta hay que prestar especial atención a los valores mostrados a continuación, ya que son los que se utilizarán para recuperar la información original y comprobar que no ha cambiado.

- *HASH*: c2ac884de2eabe95ab139f1257ba3171c037c1c9a5984a95ff8c1c1d6c819e43
- *DATOS*: https://ipfs.io/ipfs/QmdLRBtxHNm2CirRdczEc4vSrN5XrGjNWvLbwcTNfkPTUf
- *CLAVEPRIVADA*: https://ipfs.io/ipfs/QmQhk3wXrxdXrT8jKecV8y7x78KFnpio313jYpNNmoeg

Como intento de eliminar las pruebas denunciadas, se procede a borrar el mensaje ofensivo de la segunda cuenta de Twitter como puede verse en la Figura 6.7.

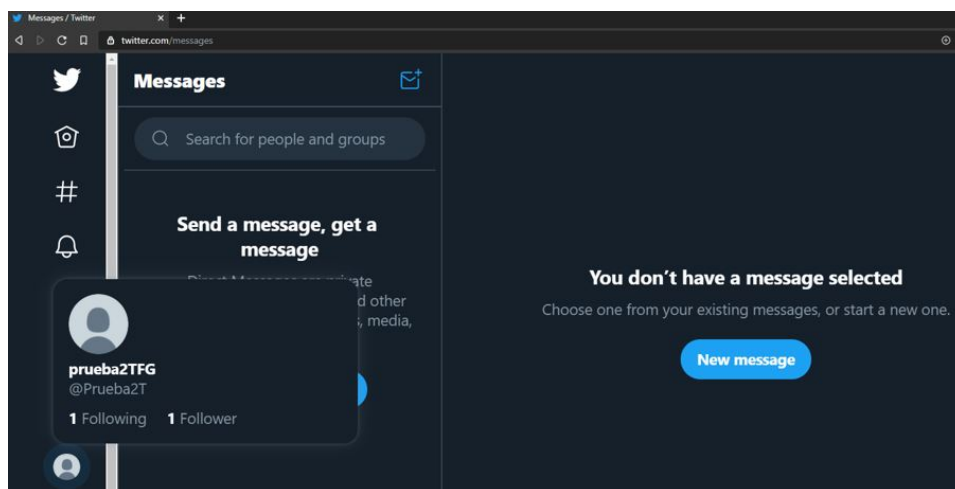


Figura 6.7: Eliminación de pruebas

Para recuperar los datos denunciados, se recurre al fichero *COMPROBACIÓN.js* y se introducen los dos enlaces de IPFS junto con la palabra aleatoria con la que se cifró la clave privada. Se ha de aclarar que para la descarga de los datos de IPFS se necesita tener el *daemon* activado por lo que se debe ejecutar el comando visto en el Listado de código 6.1.

```
1 $:\> ipfs daemon
```

Listado de código 6.1: Inicialización del daemon de IPFS

En el Listado de código 6.2 se muestra la introducción de dichos datos en el fichero *COMPROBACIÓN.js*

```
1 const DATOS = 'https://ipfs.io/ipfs/
  QmdLRBtxHNm2CirRdczEc4vSrN5XrGjNWvLbwcTNfkPTUf'
2
3 const CLAVEPRIVADA = 'https://ipfs.io/ipfs/
  QmQhk3wXrxdXrT8jKecV8y7x78KFnpio313jYpNNmoeg'
4
5 const STRING = 'TFG'
6
7 main(DATOS, CLAVEPRIVADA, STRING)
```

Listado de código 6.2: Inicialización del daemon de IPFS

Para acabar este capítulo en las Figuras 6.8 y 6.9 se ilustran el contenido **HTML** y el **hash** devueltos por el fichero siendo este último **igual** al almacenado en *Ganache*.

```

28 const DATOS = 'http://localhost:8080/ipfs/QmdLRBtxHNm2C1rRdczEc4vSrN5XrGjNwvLbwcTNfkPTUf'
29 const CLAVEPRIVADA = 'http://localhost:8080/ipfs/QmQhk3wXrxdXrT8jKecV8y7x78KFnpito313jYpNNmoeg'
30 const STRING = 'TFG'
31 main(DATOS, CLAVEPRIVADA, STRING)
32

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```

llying_Month_Emoji_V2\\Anti_Bullying_Month_Emoji_V2.png&quot;;&quot;chefbenitomolina&quot;;&quot;https:\\\\\\abs.
oji_BENITO_V2.png&quot;;&quot;behindthemaalways&quot;;&quot;https:\\\\\\abs.twimg.com\\hashflags\\RemembranceDay2
uot;https:\\\\\\abs.twimg.com\\hashflags\\2019Streamy_Awards_Emoji\\2019Streamy_Awards_Emoji.png&quot;;&quot;th
MLS_19_TFC.png&quot;;&quot;dellnorock2019&quot;;&quot;https:\\\\\\abs.twimg.com\\hashflags\\Rock_in_Rio_DellBR_F
/\\\\abs.twimg.com\\hashflags\\Nespresso_Sustainability_Emoji_V2\\Nespresso_Sustainability_Emoji_V2.png&quot;;&quot;
ji_FestivalDiaDeMuertosCDMX\\Emoji_FestivalDiaDeMuertosCDMX.png&quot;;&quot;playlikeanewyorker&quot;;&quot;https:
am_Emojis_2019_2020_NyRangers.png&quot;;&quot;maleficentmächtederfinsternis&quot;;&quot;https:\\\\\\abs.twimg.com\\
cent_Emoji_V3.png&quot;;&quot;maddennfl&quot;;&quot;https:\\\\\\abs.twimg.com\\hashflags\\Madden20_Custom_Emoji\\
tps:\\\\\\abs.twimg.com\\hashflags\\DisneyStarWarsROSLightSaber_Emoji\\DisneyStarWarsROSLightSaber_Emoji.png&qu
Luigi_Mansion3_Custom_Emoji\\Nintendo_Luigi_Mansion3_Custom_Emoji.png&quot;;&quot;blizzcon&quot;;&quot;https:\\\\\\
;,&quot;mögediemachtmitdirsein&quot;;&quot;https:\\\\\\abs.twimg.com\\hashflags\\DisneyStarWarsROSLightSaber_Emo
uot;https:\\\\\\abs.twimg.com\\hashflags\\TheOuterWorldsLaunch_Emoji_V2\\TheOuterWorldsLaunch_Emoji_V2.png&quot
/adidas_FridayNightStripes_Custom_Emoji_LightBlueBall\\adidas_FridayNightStripes_Custom_Emoji_LightBlueBall.png&c
L_Clubs_2019_2020_Emojis_CarolinaPanthers\\NFL_Clubs_2019_2020_Emojis_CarolinaPanthers.png&quot;;&quot;readywilli
V2_2019_Emojis_Mouz\\Esports_All_Access_V2_2019_Emojis_Mouz.png&quot;;&quot;magickingdom&quot;;&quot;https:\\\\\\
yEars_Flight2.png&quot;;&quot;,&quot;initialState&quot;;&quot;title&quot;;&quot;Iniciar sesi\\u00f3n en Twitter&quot;;&quot;
ot;cache_ttl&quot;;300,&quot;body_class_names&quot;;&quot;three-col logged-out western es&quot;;&quot;doc_class_na
t;login&quot;;&quot;page_container_class_names&quot;;&quot;AppContent wrapper wrapper-login&quot;;&quot;ttft navig
le" value="app/pages/login">\n <input type="hidden" id="swift-module-path" value="https://abs.twimg.com/k/swift/e
30213.js" async></script>\n\n </body>\n</html>\n',
c2ac884de2eabe95ab139f1257ba3171c037c1c9a5984a95ff8c1c1d6c819e43
PS C:\Users\salmi\TFG\SRC>

```

Figura 6.8: Respuesta del fichero *COMPROBACIÓN.js*

```

c2ac884de2eabe95ab139f1257ba3171c037c1c9a5984a95ff8c1c1d6c819e43
PS C:\Users\salmi\TFG\SRC>

```

Figura 6.9: Ampliación del *hash* devuelto

Capítulo 7

Conclusiones

En este proyecto se ha propuesto una mejora para un sistema de almacenamiento de evidencias con el que se pretende dar una solución ante la eliminación o modificación de las pruebas denunciadas. Todo esto gracias a tecnologías como Blockchain e IPFS que permiten guardar datos de forma distribuida y descentralizada, junto con el uso de protocolos de encriptación para la protección de la información.

En el desarrollo del trabajo de fin de grado se comenzó explicando los objetivos perseguidos así como los contenidos teóricos necesarios para la comprensión del proyecto.

Posteriormente se habló del problema a la hora de almacenar las pruebas denunciadas llegando a la conclusión de utilizar un cifrado de la información para la subida de los datos a IPFS.

A continuación se expuso mediante diagramas y explicaciones del código el funcionamiento del proyecto junto con un ejemplo real basado en dos cuentas de *Twitter*.

Por último, en el manual de usuario mostrado al final de la memoria se explican todas las pautas que hay que seguir para la utilización del sistema completo.

7.1. Valoración personal

Repecto a mi valoración personal de este proyecto decir que me ha ayudado a entender mucho mejor cómo funciona tanto *Blokchain* como IPFS al igual que he aprendido conocimientos básicos de programación asíncrona y orientada a objetos como es el lenguaje *Javascript*. Todo esto sumado a la utilización de herramientas de Ethereum que; personalmente, creo que tiene un gran potencial y aporta otras funcionalidades a la cadena de bloques más allá del intercambio de *criptodivisas*.

Confío en que sigan desarrollándose aplicaciones basadas en estas tecnologías y aumente el personal dedicado a este sector para proponer sistemas

sólidos de gestión y almacenamiento de datos.

Desde mi punto de vista se ha exagerado la publicidad negativa hacia *Blockchain* ya que es una tecnología en desarrollo que puede presentar una amplia gama de soluciones ante problemas de corrupción y uso inadecuado de la información. Creo que el mal uso que se da a una tecnología no debe condicionar su progreso, si no más bien, incentivar a los ingenieros y al personal especializado en el sector a seguir implementando nuevos sistemas que dificulten las actividades malintencionadas.

Pienso que se debería concienciar más a los usuarios y especialmente a los jóvenes sobre el daño que ocasiona el mal uso de las redes sociales y preocuparnos; yo el primero, por el regalo que hacemos de nuestros datos a cualquier aplicación o plataforma ya que estoy seguro que tienen una repercusión muy importante la cuál la mayoría desconocemos.

Para acabar, he de decir que estoy muy contento con el trabajo realizado debido a la formación adquirida en una temática que me atrae bastante y creo que puede desempeñar un papel importante en un futuro cercano.

Bibliografía

- [1] F. R. Más and A. D. Rosado, “La informática forense: el rastro digital del crimen,” *Derecho y Cambio Social*, vol. 8, no. 25, p. 21, 2011.
- [2] S. Nakamoto *et al.*, “Bitcoin: A peer-to-peer electronic cash system,” 2008.
- [3] A. M. Antonopoulos, *Mastering Bitcoin: unlocking digital cryptocurrencies*. O’Reilly Media, Inc., 2014.
- [4] M. B. Taylor, “Bitcoin and the age of bespoke silicon,” in *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. IEEE, 2013, pp. 1–10.
- [5] A. M. Antonopoulos and G. Wood, *Mastering ethereum: building smart contracts and dapps*. O’Reilly Media, 2018.
- [6] C. Dannen, “Solidity programming,” in *Introducing Ethereum and Solidity*. Springer, 2017, pp. 69–88.
- [7] G. Wood *et al.*, “Ethereum: A secure decentralised generalised transaction ledger,” *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.
- [8] W.-M. Lee, “Testing smart contracts using ganache,” in *Beginning Ethereum Smart Contracts Programming*. Springer, 2019, pp. 147–167.
- [9] P. Hartel and M. van Staalduinen, “Truffle tests for free-replaying ethereum smart contracts for transparency,” *arXiv preprint arXiv:1907.09208*, 2019.
- [10] W.-M. Lee, “Using the metamask chrome extension,” in *Beginning Ethereum Smart Contracts Programming*. Springer, 2019, pp. 93–126.
- [11] J. Benet, “Ipfs-content addressed, versioned, p2p file system,” *arXiv preprint arXiv:1407.3561*, 2014.
- [12] S. Chacon and B. Straub, *Pro git*. Apress, 2014.

- [13] B. Cohen, “Incentives build robustness in bittorrent,” in *Workshop on Economics of Peer-to-Peer systems*, vol. 6, 2003, pp. 68–72.
- [14] P. Maymounkov and D. Mazieres, “Kademlia: A peer-to-peer information system based on the xor metric,” in *International Workshop on Peer-to-Peer Systems*. Springer, 2002, pp. 53–65.
- [15] M. J. Freedman, E. Freudenthal, and D. Mazieres, “Democratizing content publication with coral,” in *NSDI*, vol. 4, 2004, pp. 18–18.
- [16] I. Baumgart and S. Mies, “S/kademlia: A practicable approach towards secure key-based routing,” in *2007 International Conference on Parallel and Distributed Systems*. IEEE, 2007, pp. 1–8.
- [17] D. D. F. Mazières, “Self-certifying file system,” Ph.D. dissertation, Massachusetts Institute of Technology, 2000.

Siglas

BTC Bitcoin.

DHT Distributed Hash Table.

DNS Domain Name System.

DSHT Distributed Sloopy Hash Table.

ETH Ether.

EVM Ethereum Virtual Machine.

IPFS Interplanetary File System.

IPNS Interplanetary Name System.

PoA Proof of Authority.

PoS Proof of Stake.

PoW Proof of Work.

SFS Self-Certified File System.

VCS Version Control System.

Apéndice A

Manual de usuario

Lo primero es descargar el archivo que contiene los ficheros descritos en el proyecto. Para ello se debe tener instalado **git** e introducir en una terminal el comando: ***git clone <https://github.com/Salvamr96/TFG.git>***

A.1. Requisitos

- Sistema operativo: ***Windows 10 Home***.
- Navegador:
 - *Firefox*
 - *Google Chrome*
 - *Brave*
 - *Opera*
- Tener instalada la extensión ***Metamask*** en el navegador.
- Instalar ***NodeJS***: <https://nodejs.org/es/download/>
- Instalar herramientas utilizadas:
 - ***Truffle***: <https://github.com/trufflesuite/truffle>
 - ***Ganache***: <https://github.com/trufflesuite/ganache>

A.2. Instalación de paquetes necesarios

Desde una terminal se procede a la instalación de los paquetes necesarios en el proyecto.

- `$:\> npm install -g go-ipfs`
- `$:\TFG> npm install request`

- `$:\TFG> npm install js-sha256`
- `$:\TFG> npm install web3@0.20.7`
- `$:\TFG> npm install web3-providers-http`
- `$:\TFG> npm install comment-json`
- `$:\TFG> npm install express`
- `$:\TFG> npm install body-parser`
- `$:\TFG> npm install ipfs-http-client`
- `$:\TFG> npm install openpgp`

A.3. Pasos a seguir

Una vez instaladas todas las dependencias, se debe abrir una terminal en modo administrador y configurar el nodo IPFS. Para ello ejecutar los comandos mostrados a continuación:

- `$:\> mkdir C:\SPB_Data\.ipfs`
- `$:\> ipfs init`
- `$:\> ipfs daemon`

La terminal con el *daemon* de IPFS activado se deberá dejar ejecutandose en segundo plano durante la ejecución del código. A continuación desde otra terminal se debe entrar en el directorio *SRC* y ejecutar el fichero *recompile.sh* para compilar el proyecto.

Seguidamente se debe iniciar **Ganache**. Para ello, acceder al directorio donde se haya descargado el repositorio y ejecutar el comando:

- `$:\ganache> npm install`
- `$:\ganache> npm start`

Esto abrirá la interfaz donde se debe crear un nuevo espacio de trabajo. Para ello, entrar en *new workspace* e introducir el nombre deseado para el espacio de trabajo y lo más importante: añadir el fichero **truffle-config.js** contenido en el directorio SRC. Debe quedar como en la Figura A.1

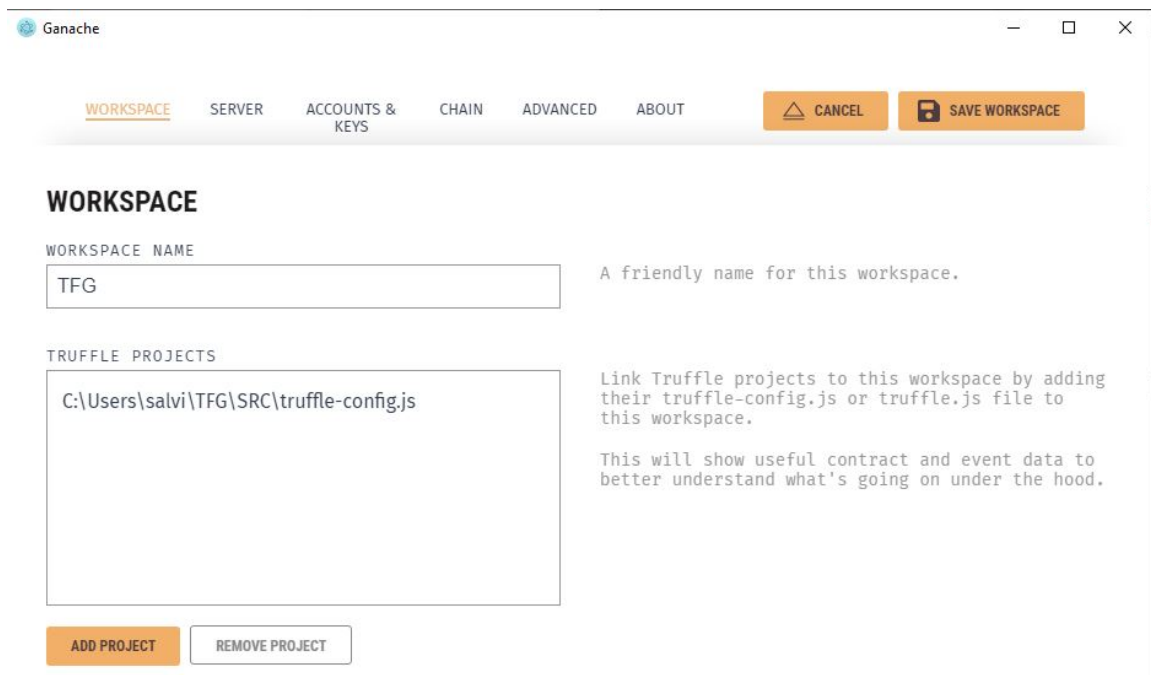


Figura A.1: Configurando Ganache

Siguiendo con la configuración, en el servidor debe dejarse los valores que vienen por defecto excepto el puerto usado que debe ser el **8545** como puede verse en la Figura A.2.

Figura A.2: Configurando Ganache

En este punto se guarda la configuración del espacio de trabajo y se podrán observar las 10 cuentas que *Ganache* pone a disposición del usuario cada una con 100 ETHs. En la Figura A.3 se muestran las dos primeras ya que son las que se utilizan en el código del proyecto.

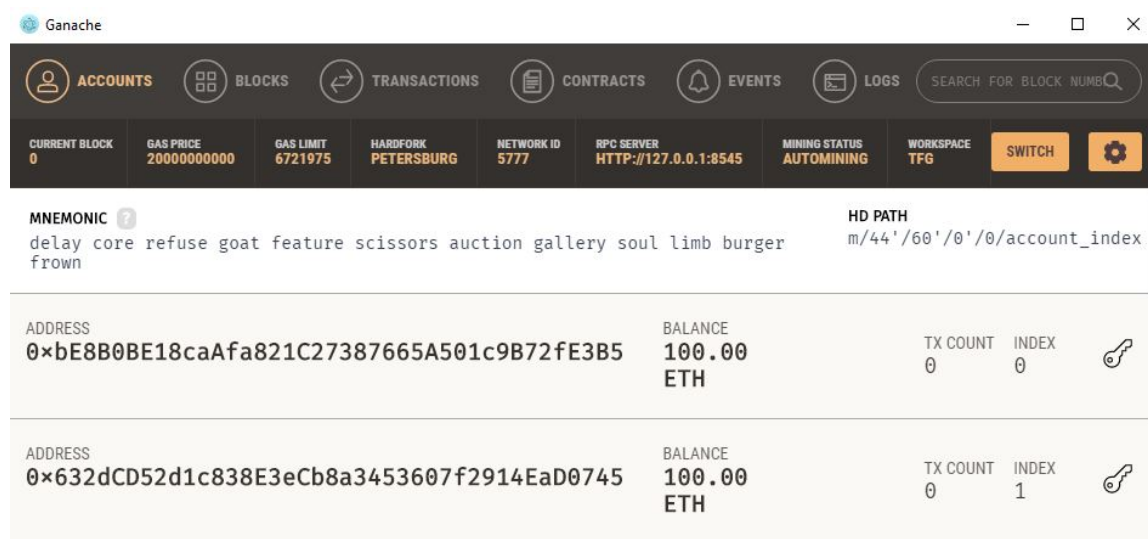


Figura A.3: Configurando Ganache

Además se observa el **MNEMONIC** que se debe introducir en **Meta-mask** para que gestione el pago ejecutado del *Cliente* hacia el *Oráculo*. Por lo tanto, el siguiente paso es acceder al navegador y en la extensión de *Metamask* introducir esta cadena de caracteres. Para ello se tiene que acceder a la importación de una cuenta mediante la semilla que se muestra en la parte inferior de la ventana de *Metamask* e introducir dicha frase como se representa en la Figura A.4.



Figura A.4: Importación de la cuenta en Metamask

Ahora se deben compilar los contratos con **Truffle** para que se guarden en *Ganache*, con lo que se procede a abrir una tercera terminal y ejecutar los comandos:

- `$:\SRC> truffle developer`
- `truffle(develop) > migrate`

Una vez compilados los contratos, se accede a Ganache para comprobar que se han cargado correctamente como en la Figura A.5.

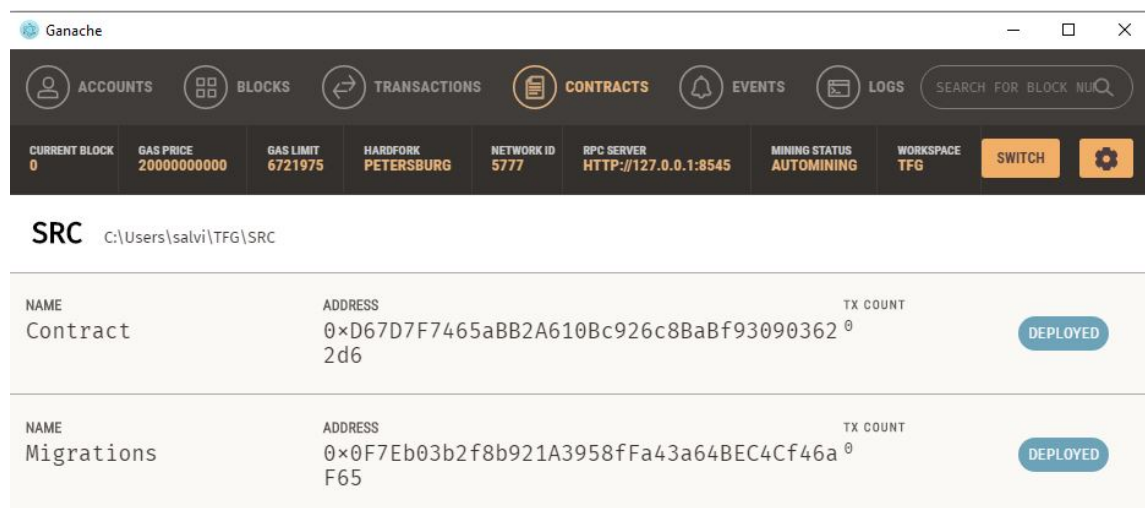


Figura A.5: Dirección de los contratos

Notar que es muy importante introducir la **dirección del contrato** en el fichero *ethereum.js* que se encuentra en el directorio *SRC* y la **dirección de la segunda cuenta** de *Ganache* en el fichero *index.html* alojada en el directorio *SRC\server\public*.

Con todo configurado falta introducir la **palabra aleatoria** con la que se quiere cifrar la clave privada en el fichero *Oracle.js* y lanzar la aplicación con el comando:

- `$:\SRC\server> node app.js`

Una vez hecho esto, acceder a la dirección `localhost:8080` en el navegador y se mostrará la interfaz *web* vista en la Figura A.6.

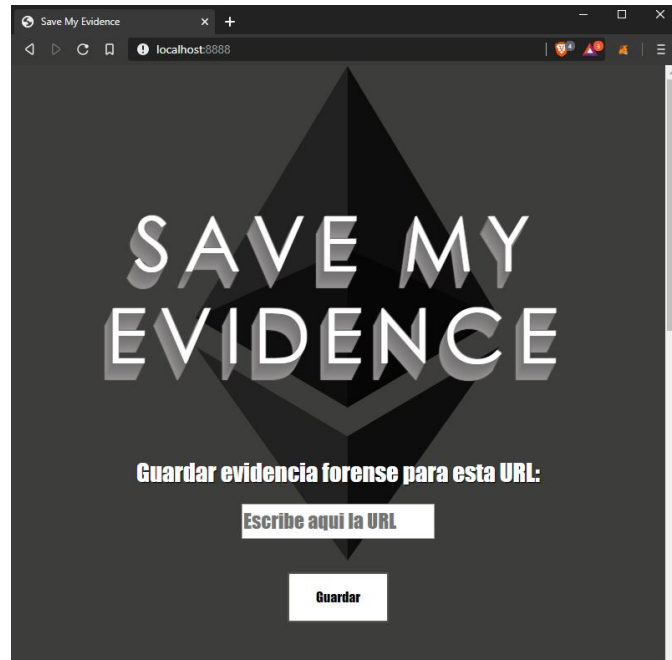


Figura A.6: Interfaz web

A continuación se introduce la URL deseada y se confirma el pago en la ventana emergente de *Metamask* mostrada en la Figura A.7. Por tanto el saldo de las cuentas queda como en la Figura A.8

Por último ya se puede acceder a Ganache para ver los resultados mostrados en las Figuras A.9, A.10, A.11, A.12.

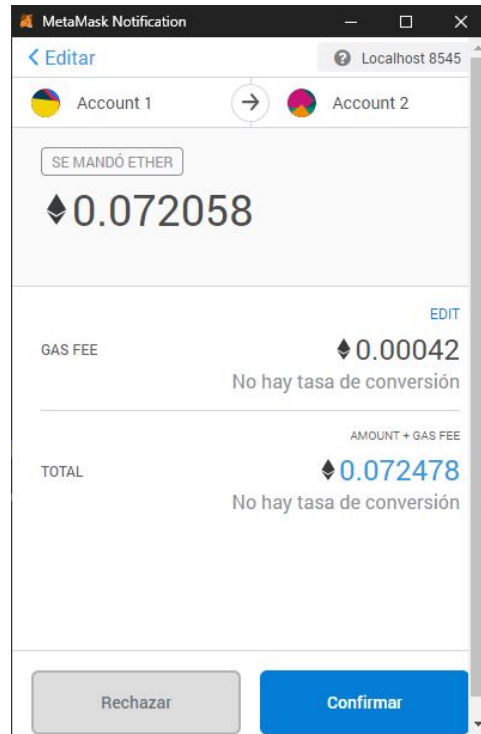


Figura A.7: Confirmación de pago en Metamask

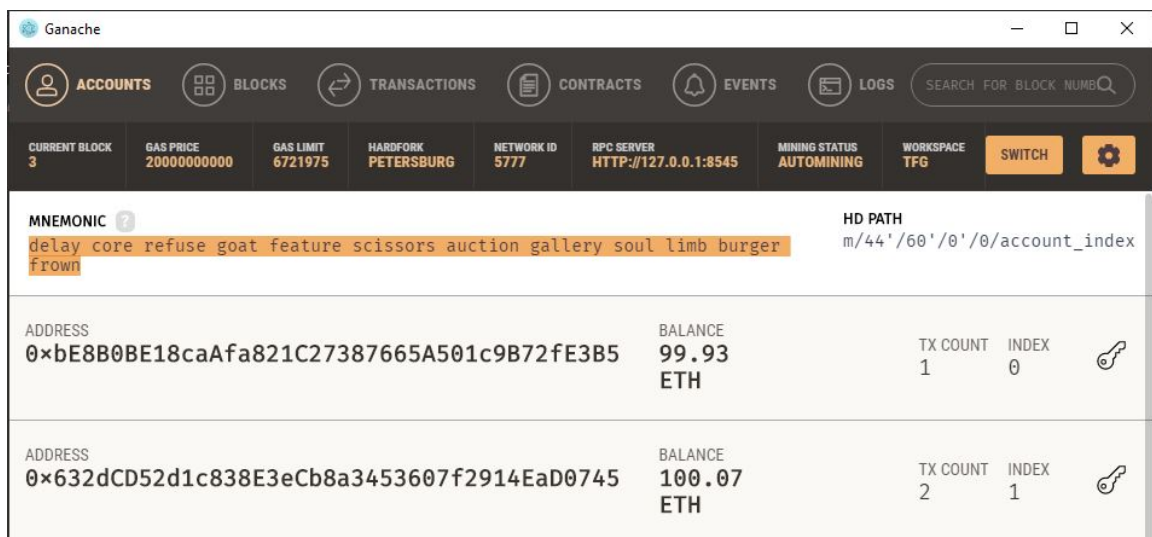







Figura A.8: Saldo de las cuentas cuando se ejecuta el pago


 ACCOUNTS

 BLOCKS

 TRANSACTIONS

 CONTRACTS

 EVENTS

 LOGS

SEARCH FOR BLOCK NUMBER

CURRENT BLOCK
3

GAS PRICE
20000000000

GAS LIMIT
6721975

HARDFORK
PETERSBURG


NETWORK ID
5777

RPC SERVER
HTTP://127.0.0.1:8545

MINING STATUS
AUTOMINING

WORKSPACE
TFG

SWITCH



BLOCK
3

MINED ON
2019-11-01 02:55:11

GAS USED
21000

1 TRANSACTION

BLOCK
2

MINED ON
2019-11-01 02:54:47

GAS USED
44312

1 TRANSACTION

BLOCK
1

MINED ON
2019-11-01 02:54:42

GAS USED
23512

1 TRANSACTION

Figura A.9: Bloques minados

CONTRACT

CONTRACT Contract	ADDRESS 0xD67D7F7465aBB2A610Bc926c8BaBf930903622d6
FUNCTION solicitarDatos(URL_solicitada: string, currentDate: uint256)	
INPUTS http://etsiit.ugr.es/, 1572573282	

Figura A.10: Bloque que contiene los datos de la función *solicitarDatos()*

CONTRACT

CONTRACT Contract	ADDRESS 0xD67D7F7465aBB2A610Bc926c8BaBf930903622d6
FUNCTION guardarSolicitud(timestamp: uint256, datosJSON: string, version: uint128)	
INPUTS 1572573282, {"WEB":"ETS de Ingenierías Informática y de Telecomunicación > Presen tación","HASH":"6b92f86a296027b3bfee2e42b732df62e88a3ab363509a9ff12cd52f8c1dca05","D ATOS":"https://ipfs.io/ipfs/Qmbbra6qwo4RiVVUE4c49kMaFqmgRtqjS3Va6NZGFn8mJG","CLAVEPR IVADA":"https://ipfs.io/ipfs/QmeSV8LP4FxUdSYA33bhLsVnXSHb6ciAcpRKgUZRUy55Jh"}, 1	

Figura A.11: Bloque que contiene los datos de la función *guardarSolicitud()*

← BACK		TX		
		0×fb2be859363a129dc81c8708dd1616eeb99035193c5d7de91bce3388fbf1ab5d		
SENDER ADDRESS		TO CONTRACT ADDRESS		CONTRACT CALL
0×bE8B0BE18caAfa821C27387665A501c9B72fE3B5		0×632dCD52d1c838E3eCb8a3453607f2914EaD0745		
VALUE	GAS USED	GAS PRICE	GAS LIMIT	MINED IN BLOCK
0.07 ETH	21000	20000000000	21000	3
TX DATA				
0x				

Figura A.12: Bloque que contiene el pago del Cliente al Oráculo