TRABAJO FINAL DE GRADO

TÍTULO DEL TFG: Explorando la Blockchain de Ethereum y el desarrollo de smart contracts

TITULACIÓNES: Grado en Ingeniería de Sistemas de Telecomunicaciones, Grado en Ingeniería Telemática

AUTOR: Víctor Miranda Palacios

DIRECTOR: Juan Bautista Hernández Serrano

FECHA: 6 de febrero del 2018

Título: Explorando la Blockchain de Ethereum y el desarrollo de smart contracts

Autor: Víctor Miranda Palacios

Director: Juan Bautista Hernández Serrano

Fecha: 6 de febrero del 2018

Resumen

En este proyecto se pretende dar a conocer la tecnología Blockchain, ver cómo surgió, cómo funciona y el motivo por el cual posiblemente sea uno de las grandes tecnologías del futuro.

El Blockchain, a grandes rasgos, es una gran base de datos descentralizada y en base a algunas de las cualidades descritas durante el proyecto, se creará una prueba de concepto basada en Ethereum, una de las blockchains más conocidas e importantes después de Bitcoin, con el fin de llevar la protección de datos, en especial los datos médicos, a un siguiente nivel.

Para permitir descentralizar la información médica y dotar de una amplia privacidad de nuestros datos, se desarrollará una aplicación web que extraerá los datos de la blockchain. Se usarán *smart contracts,* que son pequeños scripts, que son almacenados en la blockchain de Ethereum y como si de contratos reales se tratarán, permiten cumplir al pie de la letra las normas y reglas descritas en ellos. La aplicación permitirá que pacientes y doctores compartan información clínica sin que esta esté almacenada de manera centralizada por hospitales o médicos privados. El objetivo es bloquear el acceso a nuestros datos, hasta que haya un consentimiento por nuestra parte; no dar pie a posibles filtraciones de historiales o datos médicos valiosos de pacientes.

El proyecto se ha diseñado usando los lenguajes de programación Solidity y Javascript. Durante el desarrollo de la aplicación se ha usado el sistema operativo Ubuntu 16.04LTS y para el despliegue de los contratos en Ethereum se ha utilizado una red test para abaratar costes y disminuir el tiempo que la blockchain tarda en comprobar cada una de las transacciones.

Title: Exploring the Ethereum Blockchain and the development of smart contracts

Author: Víctor Miranda Palacios

Director: Juan Bautista Hernández Serrano

Date: February 6th 2018

Overview

In this project we try to put light on the Blockchain: how it emerged, how it works and why it is possibly one of the greatest technologies of the future.

The blockchain, in big terms it's a big decentralized database, and based on these qualities described during the project, a proof of concept will be created based on Ethereum, one of the most known and important blockchains after Bitcoin, to take data protection, especially medical data, to a next level.

To decentralize medical information and provide wide privacy to our data, it will develop a web application that will take the data from the blockchain. Smart contracts will be used are basically small scripts, that are stored in the Ethereum's blockchain. As if they were real contracts, it allows to comply accurate the rules that are described in them. The application will allow patients and doctors to share clinical information without it being stored centrally by hospitals or private doctors. The goal is to block all access to our data without prior consent. Do not give rise to possible leaks of patient medical records and data.

The project has been designed using the Solidity and Javascript programming languages. During the development, it was used the operating system Ubuntu 16.04LTS and for the deployment of the Ethereum contracts, the test network has been used to reduce costs and reduce the time it takes for the blockchain to check each of the transaction.

It has been achieved in this way, that this and other projects take advantage of the capabilities of technology in a simple way, allowing the owner of the data to be the final user.

Agradecimientos

En primer lugar deseo expresar mi agradecimiento al director de este trabajo final de grado, Juan Bautista Hernández Serrano, por su esfuerzo y dedicación en asignaturas donde hemos coincido durante mi etapa académica y donde gracias a su clases impartidas en materia de seguridad informática, me ha abierto una posible futura vocación.

Asimismo, agradezco también a José Luis Muñoz Tapia, por su constante apoyo durante la realización de este trabajo. Por ofrecerme todo sus conocimientos en la materia, darme ideas y enfocarme el proyecto. Su seminario de introducción a la tecnología Blockchain, me permitió conocer un montón de conceptos para empezar a investigar y desarrollar el proyecto. Agradezco muchísimo todo el tiempo que ha invertido en mí.

Gracias a mi familia, a mi novia y la familia de ella, por la plena confianza en mí y su apoyo incondicional.

A todos, muchas gracias.

ÍNDEX

INTRODUCCIÓN 1

CAPÍTULO 1. LA TECNOLOGÍA BLOCKCHAIN 3

1.1. El nacimiento de una nueva tecnología 3

1.2. El Bitcoin y la blockchain 4

CAPÍTULO 2. LOS SMART CONTRACTS Y ETHEREUM 8

2.1. Ethereum 9

2.2. Ethereum Virtual Machine 10
2.2.1. El gas 12

CAPÍTULO 3. DESARROLLANDO SMART CONTRACTS 14

3.1. El entorno de trabajo - Remix 14

3.2. Solidity 16
3.2.1. Diseño de los ficheros 17
3.2.2. Tipos de datos 18
3.2.3. Unidades y variables especiales disponibles 20
3.2.4. Estructura interior de los contratos 21

3.3. Desarrollo de los *smart contracts* de Health Hub 29

CAPÍTULO 4. APLICACIONES DESCENTRALIZADAS 32

4.1. Desarrollando la Đapp 33
4.1.1. Instalación y configuración 33

CAPÍTULO 5. UN FUTURO PROMETEDOR 42

5.1. Blockchains a las que seguir el paso 43

5.2. Nueva tecnología más allá del blockchain 44

5.3. El futuro de Ethereum 45

BIBLIOGRAFÍA 46

ANEXOS 50

1. Contratos de Ejemplo 50
1.1. Hello World 50
1.2. My Hello World 50
1.3. Depósito de ether 51
1.4. Interacción entre contratos 52
1.5. El Concierto 53

1.6. El Banco 54

2. Contratos de Health Hub 55
2.1. Contrato de los Doctores (Doctor.sol) 56
2.2. Contrato de los Pacientes (Patient.sol) 57
2.3. Contrato de la información de Pacientes (Info.sol) 62

INTRODUCCIÓN

La tecnología moderna permite a las personas comunicarse directamente. Llamadas de voz y vídeo, mensajes de correo electrónico y mensajes instantáneos viajan directamente de A hasta B, manteniendo la confianza entre las personas sin importar lo de lejos que estén. Cuando se trata de dinero, la gente tiene que confiar en que una tercera parte pueda completar una transacción, como sería el caso de los bancos.

La tecnología Blockchain está desafiando el statu quo de una manera radical mediante el uso de las matemáticas y la criptografía. Blockchain proporciona un proceso descentralizado y abierto de cada transacción, por lo tanto, con la criptomoneda Bitcoin se deja de lado el poder de los bancos, para otorgar el poder a los usuarios. Bitcoin rompe las barreras de la centralidad de los servicios hacia un mundo descentralizado.

Pero no solo el dinero. Las blockchains pueden integrar bienes, propiedades, trabajo o incluso votos, creando un registro cuya autenticidad puede ser verificada por toda la comunidad.

Aprovechando esta nueva tecnología que en los últimos años ha sufrido un gran auge gracias a que cada vez es más conocida por la población, han nacido múltiples criptomonedas y múltiples blockchains que abarcan diferentes servicios. Ethereum es una de ellas y gracias a la gran comunidad de desarrolladores que tiene, permitirá que se convierta en una de las blockchains más importantes en los próximos años.

Enfocándonos en la privacidad de los datos, es evidente que la información es apreciada por muchos aspectos relevantes. Por ejemplo, en el ámbito empresarial su importancia radica en la utilidad de los datos para la toma de decisiones importantes. El problema llega cuando consideramos que todos los datos recopilados pueden pertenecer a los clientes o usuarios. Por ello, en los últimos años ha cobrado relevancia la protección de los datos personales.

Debido a la importancia de la información sensible y personal de los usuarios y a los beneficios que pueden generarle tanto a grandes multinacionales como a cibercriminales, que buscan adueñarse de estos datos, continuamente se observan noticias relacionadas con la fuga de información, en los cuales se utilizan distintos vectores de ataque para lograrlo y es importante poner medidas.

Unos de los datos más importantes y que se están llevando a la era digital, son los datos médicos. Con el incremento de dispositivos que obtienen en tiempo real datos biométricos, han hecho incrementar el número de datos que se usarán para fines médicos drásticamente en la última década. Realizar el seguimiento del historial médico de un paciente (información de visitas pasadas, resultados de laboratorio, medicamentos, mediciones en tiempo real de las constantes o alergias a fármacos) mediante un sistema tradicional se ha convertido en una tarea bastante difícil para un médico. Y no solo eso, una vez se han recopilado todos estos datos, los usuarios han de ser capaces de consultarlos con facilidad.

Actualmente también surgen problemas cuando los datos se han de consultar fuera de nuestras fronteras. Posibles emergencias fuera de nuestro país podrían solucionarse rápidamente si tenemos a mano nuestros historiales clínicos.

Los *smart contracts* de Ethereum, pueden ser una gran oportunidad para descentralizar todos estos datos y protegerlos en la blockchain.

Para ello se va a crear una aplicación llamada **Health Hub** basada en la blockchain de Ethereum, con los *smart contracts* como base, para salvaguardar todos los datos médicos. Donde los dueños de los datos son los propios pacientes, que podrán consultar toda su información y estar seguro que será totalmente confidencial. Veamos que nos ofrece la tecnología y cómo se puede lograr esto.

CAPÍTULO 1. LA TECNOLOGÍA BLOCKCHAIN

Una noche tranquila de verano en junio de 1947, en medio de un pequeño poblado situado en Roswell, Nuevo México, se desata el principio de la historia del presunto choque de la nave extraterrestre más famosa de todos los tiempos. Sin un segundo que perder, la prensa del país se hace eco del suceso y no tardan en llegar los primeros centenares de periodistas a la localidad.

Los militares han llegado a la zona y entre un fuerte revuelo se empiezan a tomar las primeras declaraciones a todos los individuos del condado, anotando en una libreta que es exactamente lo que han visto. Imaginemos por un momento un total de 500 entrevistas a militares, vecinos y curiosos de la zona y por suerte, todos cuentan la misma historia, con idénticos detalles, “Ha aterrizado una nave extraterrestre; Los alienígenas son de color verde”. ¿Habría alguna duda de que ese suceso ha había sucedido en realidad? Este es el principio fundamental de Blockchain.

Imaginamos ahora un fichero donde se indica una transacción monetaria entre un sujeto A hacia un sujeto B, y que ese fichero pudiera estar en miles de ordenadores duplicado, con la seguridad de que nadie puede alterar esa transacción a traición para favorecerse. Pero, ¿y si sucediera? La propia red tiene el poder de confiar en la mayoría, y en cuestión de minutos, todos se sincronizarán correctamente.

Aunque uno o centenares de los miles de millones de ordenadores conectados a la Blockchain desapareciesen de la red, no pasaría nada. Esto es lo que se consigue con esta tecnología, en esencia, es un registro distribuido resistente a modificaciones y sin la necesidad de confiar en todos los miembros que la conforman, basta con confiar en la mayoría de ellos.

Falsear el aterrizaje de la nave extraterrestre, siguiendo la analogía, llevaría a un conceso difícil de realizar, equivaldría a más del 50% de mentiras por parte de los militares de la zona, que quedaría en una mentira coordinada, algo que es prácticamente imposible de realizar.

1.1. El nacimiento de una nueva tecnología

Las blockchains son increíblemente populares hoy en día, pero qué es realmente una blockchain, cómo funciona y qué problemas solventa.

Como el nombre indica, la blockchain es unas cadenas de bloques que contienen información. Esta técnica fue descrita originariamente en el año 1991, bajo el nombre de *Digital timestamps* [1] por un grupo de investigadores. El documento ofrecía un proceso para marcar los documentos digitales de manera que no fuera posible actualizarlos o manipularlos como si de un notario se tratara.

Sin embargo, pasó bastante desapercibido sin usarse hasta que en 2009, uno o varios desarrolladores bajo el pseudónimo de **Satoshi Nakamoto**, crearon la primera criptomoneda digital bajo el nombre **Bitcoin** (BTC) [2].

La tecnología blockchain, también conocida como libro de cuentas distribuida (*Distributed Ledger*), es una base de datos distribuida por toda la red que es completamente abierta a todo el mundo. Esta base de datos tiene una característica interesante y es que cuando se ingresan nuevos datos dentro de la blockchain se vuelve extremadamente difícil modificarlos. Veamos cómo funciona.

1.2. El Bitcoin y la blockchain

Bitcoin es la criptomoneda por excelencia, tan conocida o más que la propia tecnología. Durante los últimos años, gracias a diferentes desarrolladores han ido surgiendo nuevas criptomonedas basadas en la tecnología blockchain, mejorando y modificando ciertos aspectos. Por ello es esencial conocer cómo funciona Bitcoin.

Echemos un ojo más cerca de la tecnología, y observemos la estructura de los bloques. Cada bloque contiene datos, el hash del bloque y el hash del anterior bloque **(Fig.1.1)**. La información contenida dentro de cada bloque dependerá del tipo de blockchain y los servicios que ofrezca; Bitcoin, por ejemplo, guarda el detalle de las transacciones, como es el emisor, el receptor y la cantidad de Bitcoins. El hash del bloque, puede ser comparado como si de una huella digital se tratara. Identifica el bloque en su conjunto y este identificador siempre será único.

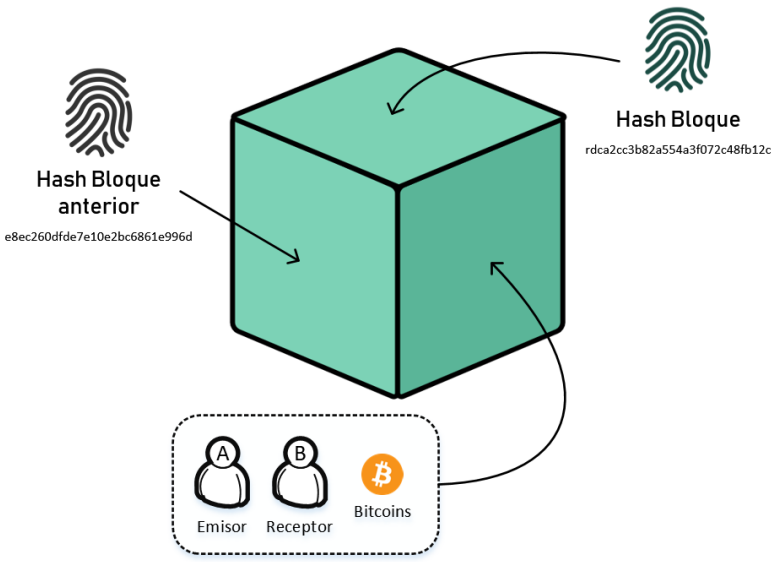

Fig. 1.1 Estructura de un bloque

Una vez se ha creado un bloque, modificar una transacción de su interior, cambiará por completo el hash. Éste es un buen mecanismo para comprobar si se ha modificado o no un bloque. El último elemento del que está compuesto un bloque es el hash del bloque anterior y esto efectivamente creará una cadena de bloques. Esta técnica hace que la blockchain sea segura, veamos el siguiente ejemplo.

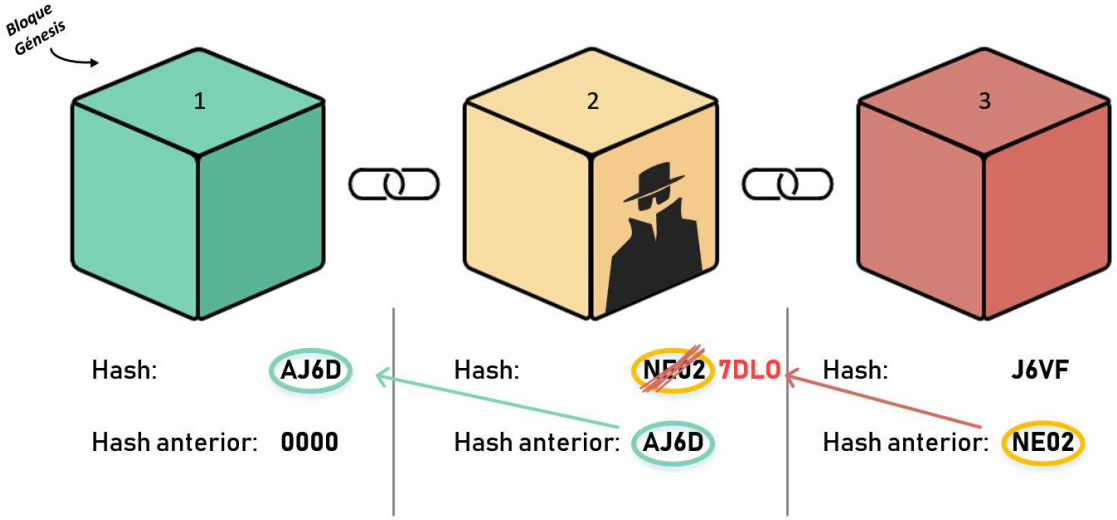

Fig. 1.2 Cadena de bloques con manipulación

Tenemos una cadena de 3 bloques (**Fig.1.2**), cada bloque tiene su hash y el hash del bloque anterior. El primer bloque, es el especial y no puede apuntar hacia un bloque pasado porque es el primero de todos y se le llama bloque Génesis. Veamos qué pasa si modificamos el segundo bloque. Esto hará que consecutivamente el hash del bloque 2 guardado sobre el bloque 3 no coincidan y por lo tanto, modificar un solo bloque, provocará que todos los siguientes queden invalidados.

El uso de hashes permite detectar, pero no prevenir la manipulación de los bloques. Hoy en día los ordenadores son suficientemente potentes como para calcular miles de hashes por segundo y alguien podría modificar un bloque y rápidamente calcular el hash del bloque modificado y los siguientes en la cadena. Para mitigar este problema, se creó el mecanismo conocido como *Proof-of-Work* (Prueba de trabajo) que también recibe el nombre de minado.

El *Proof-of-Work* permite crear una historia a base de bloques de manera que haya un consenso entre todos los mineros. Los usuarios que pretenden minar ese bloque, deberán encontrar lo más rápido posible el valor que resuelve la prueba de trabajo. Este mecanismo de consenso es muy costoso de realizar y consume muchos recursos. Al confiar en la honestidad de los mineros que encuentran la prueba de trabajo, el bloque se añade a la blockchain. Es muy fácil comprobar la validez de la prueba de trabajo, pero muy costosa encontrarla. La dificultad del mecanismo aumenta conforme más usuarios minan los bloques. Al ser un proceso tan esencial, cada vez que uno de los nodos de la red encuentre una solución válida, recibirá una recompensa de Bitcoins.

Éste es un mecanismo que ralentiza la creación de los bloques. En Bitcoin, calcular un nuevo bloque y añadirlo a la blockchain toma unos 10 minutos de media de trabajo. Manipular la blockchain es muy difícil, porque si se desea modificarla, se deberá recalcular la prueba de trabajo de los siguientes bloques.

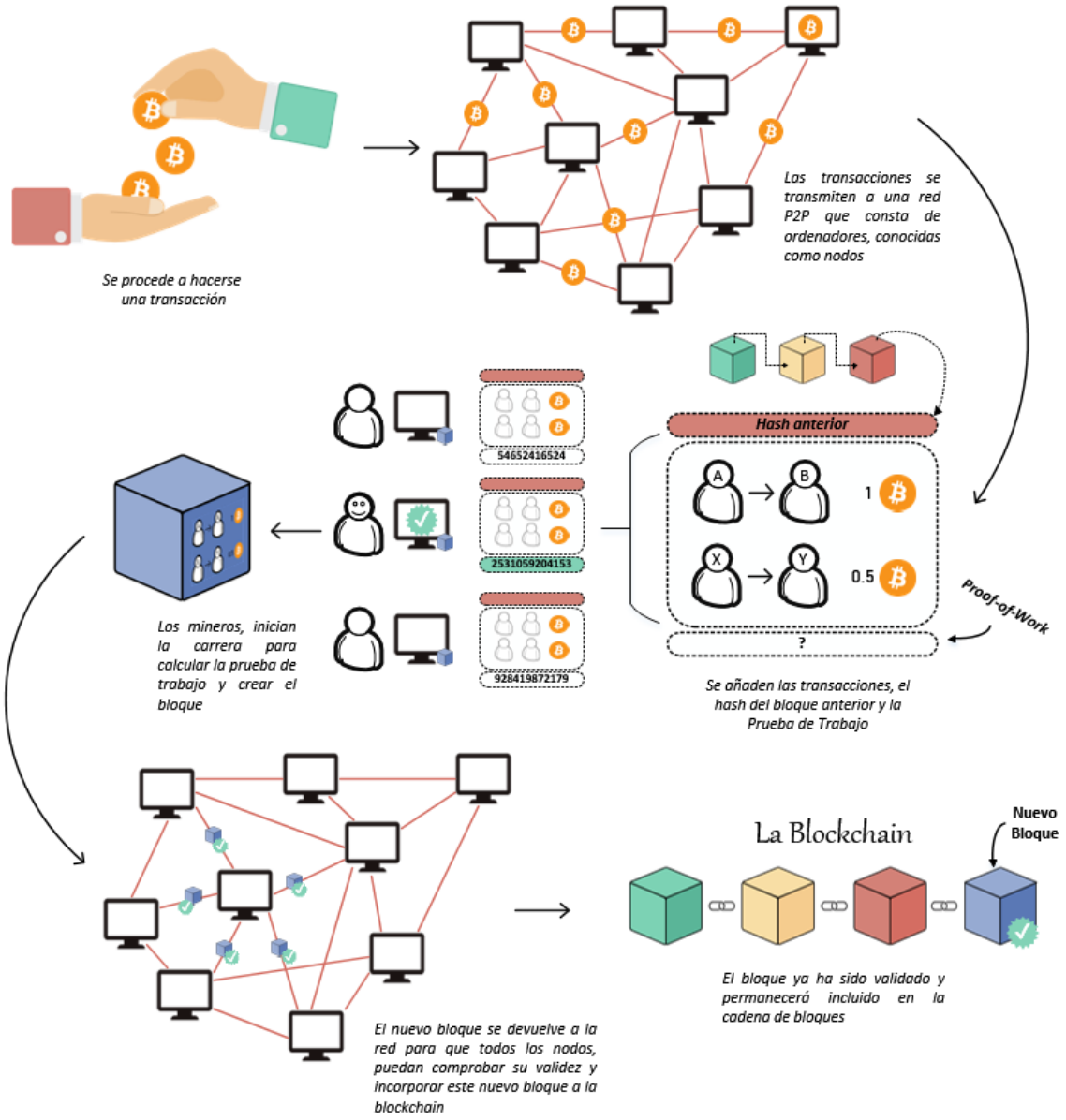

Fig. 1.3 El proceso de generación de bloques en la blockchain Bitcoin

Pero la seguridad no solo se basa en usar de manera creativa los hashes y el mecanismo de *Proof-of-Work*, sino que también, la propia red distribuida aplica seguridad a la blockchain en vez de usar una entidad central que gestione la blockchain. Blockchain usa redes peer-to-peer (P2P) donde todo el mundo tiene permisos para acceder. Cuando un nuevo usuario se une a la red, recibirá una copia de la blockchain y se convertirá en un nodo de la red. Cada vez que un

nuevo bloque sea creado, se envía y cada nodo verificará el bloque para comprobar que no ha habido manipulaciones, para posteriormente añadir el bloque a la cadena. El proceso puede verse explicado en la **(Fig.1.3)**.

Este proceso crea consenso entre todos los nodos de la red, eliminando los bloques que son manipulados. Por lo tanto, para poder manipular una blockchain, uno debería ser capaz de modificar gran parte de los bloques de la cadena, ser capaz de calcular todas las pruebas de trabajo y tomar el control de más del 50% de la red. Una hazaña imposible.

Las blockchains están en constante evolución. Uno de los recientes desarrollos en el que se aposenta este proyecto, es la creación de *smart contracts.* Estos contratos son simples programas que se guardarán en la blockchain y ejecutan un código para, por ejemplo, intercambiar monedas bajo ciertas condiciones. La nueva criptomoneda que se aprovecha de esta gran idea es Ethereum.

CAPÍTULO 2. LOS SMART CONTRACTS Y ETHEREUM

Los *smart contracts* son muy populares hoy en día. ¿Pero qué problemas solventan? El término de "smart contract" fue usado por primera vez por **Nick Szabo** [3] en 1997 mucho antes de que Bitcoin apareciese. Era un científico de la computación, experto en derecho y criptógrafo. Todas estas áreas en las que era experto, le hicieron llegar a querer crear un libro de cuentas distribuido, donde en vez de guardar transacciones, se almacenasen contratos.

Los *smart contracts* son justamente contratos como en el mundo real, pero la única diferencia es que estos son completamente digitales, de hecho, este contrato es un pequeño programa informático que es guardado dentro de una blockchain. Veamos un ejemplo para entender cómo funcionan los contratos inteligentes.

Probablemente se esté familiarizado con plataformas como las de *Kickstarter,* basadas en financiaciones colectivas, más conocidas como *crowdfunding*. Equipos de trabajo pueden ir a *Kickstarter*, crear un proyecto, establecer un objetivo de financiación y comenzar a recaudar. En esencia, esta plataforma es un intermediario entre los equipos de trabajo y los que darán apoyo. Significa que ambos tendrán que confiar en que la plataforma maneje su dinero correctamente. Si el proyecto acaba obteniendo los fondos exitosamente, el equipo esperará la recepción de esa financiación. Por otro lado, los usuarios esperarán que el dinero invertido llegue al proyecto si se ha llegado a la meta y si no, obtener de vuelta el dinero.

Ambos tienen que confiar en la plataforma, pero con *smart contracts* podemos crear un sistema similar que no requiera una tercera parte de confianza. Podemos programar un *smart contract* que mantenga todos los fondos recibidos hasta alcanzar un objetivo determinado. Una vez desplegado ese contrato, los partidarios del proyecto ahora pueden transferir su dinero al *smart contract*. SI el proyecto acaba alcanzando el objetivo, automáticamente pasará todo lo recaudado a los creadores del proyecto, sino, será devuelto.

Y al estar alojados los *smart contracts* en una blockchain, sigue las propiedades descritas al inicio; todo totalmente está distribuido. Con esta técnica, nadie tiene el control del dinero.

¿Por qué debemos confiar en estos contratos? Estos heredan de la tecnología blockchain propiedades interesantes.

- Inmutabilidad: Ser inmutable significa que una vez que se haya creado, nunca podrá ser modificado. Nadie puede malintencionadamente alterar el código del contrato.

- Distribuido: Los contratos, como pasa con las transacciones de Bitcoin, también son validados por la red.

Otras características fundamentales que tienen los *smart contracts* son:

- Deterministas: Dado que el código alojado en un *smart contract* se ejecuta simultáneamente en múltiples nodos distribuidos, debe ser determinista, es decir, dada una entrada, todos los nodos deben producir el mismo resultado. Eso implica que el código no debería tener ninguna aleatoriedad.

- Verificables: Una vez implementado un *smart contract*, obtendrá una dirección única. Antes de usarse, las partes interesadas en usarlo pueden ver y verificar el código para una mayor seguridad. Algo esencial en los contratos del mundo real.

Los usos que se le puede dar a esta idea son todos los imaginables. Creación de apuestas de todas clases, donde dos o más partes recurren a un contrato para asegurar las condiciones y los premios de la apuesta. El uso para votaciones; poder registrarse y verificarse de manera segura y exacta los resultados de cualquier votación e incluso establecer una consecuencia inmediata para los resultados. La propiedad intelectual y patentes pueden estar también sujeto a esta nueva evolución de los contratos. Y como es en el caso de este proyecto, dotar de privacidad a datos médicos.

En este momento, hay una gran variedad de blockchains que admiten *smart contracts*, pero el más grande y conocido es Ethereum. Vale la pena señalar que Bitcoin también tiene soporte para *smart contracts*, pero son muy limitados en comparación con Ethereum.

2.1. Ethereum

Ethereum es la segunda plataforma de blockchain más conocida. Fue creada por el conocido **Vitalik Buterin**, programador y escritor ruso, conocido también por ser el co-fundador de Bitcoin Magazine[1], que a finales de 2014 comenzó con el desarrollo de Ethereum.

El propósito de Ethereum es desarrollar una plataforma que permita y facilite a otros programadores la creación de aplicaciones descentralizadas con los *smart contracts* como base de estas. Al mismo tiempo sirve como plataforma mundial donde se ejecutan estas aplicaciones.

Ethereum tiene su propia criptomoneda llamada ***Ether*** (ETH), el combustible que impulsa la plataforma. Esta criptomoneda es la monedan utilizada por los clientes de la blockchain de Ethereum. Además de ser una moneda similar a Bitcoin para poder realizar pagos a otras personas, también es usado en el desarrollo de los *smart contracts*, es decir, *ether* es el incentivo que asegura que los programadores escriban aplicaciones de calidad y por último la recompensa que recibirán los mineros al ir incluyendo los bloques en la cadena.

[1] Revista del co-fundador de Ethereum: *Bitcoin Magazine* https://bitcoinmagazine.com/

Ethereum utiliza las características y la tecnología de la blockchain para crear su infraestructura para la evolución de las aplicaciones y servicios centralizados a un mundo descentralizado.

Ethereum también utiliza como Bitcoin el mecanismo de *Proof-of-Work* para la creación de nuevos bloques en la blockchain. Pero desde el año 2017 se viene hablando de que Ethereum en una de sus próximas actualizaciones cambiará su mecanismo de minado al conocido como *Proof-of-Stake* o prueba de participación.

A diferencia del *Proof-of-Work*, donde el algoritmo recompensa a los mineros que resuelven problemas matemáticos con el objetivo de validar transacciones y crear nuevos bloques, con el *Proof-of-Stake*, el creador de un nuevo bloque es elegido de manera aleatoria.

Proof-of-Stake se basa en un sorteo. Para participar en él, se debe bloquear tanta cantidad de ether como se desee. La cantidad de ether bloqueada se usará para escoger el minero del bloque. Cada ether tiene un identificador a sus espaldas y, por lo tanto, cuanto más ether bloqueado, más posibilidades hay que, como si papeletas en un sorteo se tratara, de ser elegido. Para escoger el ganador del sorteo, aleatoriamente se seleccionará usando alguno de los valores aleatorios de un bloque y no será hasta bloques más adelante que el usuario escogido mine el bloque.

En este mecanismo, no hay recompensa por minar el bloque, sino que los mineros cobran una comisión a través de cada transacción.

¿Pero por qué quieren cambiar de uno a otro? En un consenso distribuido basado en el *Proof-of-Work*, los mineros necesitan mucha energía. Según un estudio de *Digiconmist [4]* y *PowerCompare [5]*, el Bitcoin empieza a consumir más energía que 130 países del mundo.

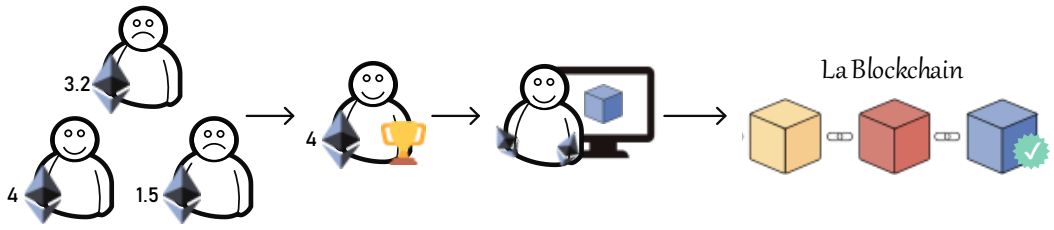

Fig. 2.1. El mecanismo de *Proof-of-Stake*

2.2. Ethereum Virtual Machine

Recordemos que el objetivo principal de Ethereum es facilitar la creación de aplicaciones que usen el blockchain, al mismo tiempo que sirve como plataforma mundial segura, de donde se nutren estas aplicaciones. Por esto, muchos

denominan a Ethereum la gran "computadora mundial" y todo esto es gracias al uso de la *Ethereum Virtual Machine* (EVM), la máquina virtual de Ethereum.

Esto nos lleva a hablar de algo un poco más técnico y es el modelo de ejecución de los *smart contracts*. Será esta EVM la que se encargará de ejecutar el código del contrato. El principal lenguaje para programar los *smart contracts* y que más adelante se explicará, es Solidity. Cuando un contrato diseñado con Solidity es compilado, se convierte en una secuencia de código de operaciones (*operation codes*) también conocidas como ***opcodes***. Los *opcodes* están referenciados por nombres mnemotécnicos como lo pueden ser ADD (para sumas) o MUL (para multiplicaciones) [6].

Los ***bytecodes*** son similares a los *opcodes*, pero representan números hexadecimales que la EVM es capaz de procesar.

Solidity →	Opcodes →	Bytecodes

`contract HelloWorld {`
`  string name;`
`  function  HelloWorld() {`
`  name = "World";`
`  }`
`  ...`
`}`
	`PUSH1 0x60`
	`PUSH1 0x40`
	`MSTORE`
	`PUSH1 0x4`
	`CALLDATASIZE`
	`LT`
	`PUSH2 0x4C`
	`...`
		`6060604052600436106`
		`1004c576000357c010000`
		`0000000000000000000`
		`0000000000000000000`
		`0000000000000000000`
		`...`

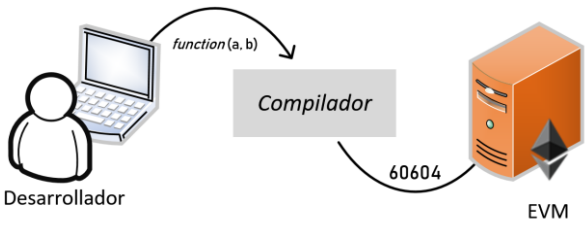

Fig. 2.2 Compilación del código en EVM

La complejidad de todo esto se centrará en los compiladores que convertirán el código de los *smart contracts* a *bytecodes* que la EVM puede entender. Mejorar el compilador, puede mejorar mucho la eficiencia del código.

Uno se puede preguntar qué puede pasar si alguien crea en un contrato con una función como esta:

```
function foo(){
     while (true){
     . . .
     }
}
```

donde un `while(true)` representa un bucle infinito. No podemos analizar el código hasta que este sea ejecutado por los nodos, y por lo tanto esto provocaría un bucle constante para siempre, un ataque de denegación de servicio (DoS). Por ello Ethereum instauró un mecanismo para protegerse de este ataque, el llamado “gas”.

2.2.1. El gas

Esta solución llamada **gas**, podemos asimilarla al fuel que necesitan las transacciones para ser ejecutadas. Cuando se habla de transacciones engloba tanto transacciones de ether de una cuenta a otra, como una ejecución de un *smart contract*.

Cada operación u *opcode* que se haga en el código tiene una cantidad de gas asignada. Por ejemplo, un ADD cuesta 3 gas, calcular un hash SHA 30 gas y enviar una transacción de ether 21.000 gas [6]. Operaciones con más dificultad computacional requerirán más gas.

El parámetro que la EVM necesita para ejecutar las transacciones es el STARTGAS, también conocido como ***transaction gaslimit*** (límite de gas). La *transaction gaslimit* es la cantidad de gas que se decide enviar en una transacción (parámetro modificable por el usuario).

En el caso de querer transferir a otro usuario ether si se indica cómo *transaction gaslimit* un total de 30000 gas, sabiendo que la transacción de ether cuesta 21000, se devolverá al usuario el gas no gastado, 9000 gas (30.000 - 21.0000). El gas también sirve como mecanismo para pagar a los mineros. En cada transacción se debe de especificar una cantidad de *transaction gaslimit* que está dispuesto a pagar, en el caso de una transacción de ether, se conoce que son 21.000 gas, pero normalmente el usuario no puede conocer cuánto gas le va a costar, por ello especifica una cantidad límite. A parte se ha de indicar cuánto se está dispuesto a pagar por cada gas. El gas solo existe dentro de la máquina virtual Ethereum. Cuando se trata de pagar por el gas consumido, la tarifa se cobra con la equivalencia en ether. A este otro termino se le conoce como ***gasprice*** (precio del gas) definido como Gwei/gas (1 Gwei es 10^9 ether).

¿Por qué las operaciones no tienen un coste medido en ether directamente? La respuesta es que el ether, como los bitcoins, tiene un precio de mercado que puede cambiar rápidamente, pero el coste de la computación no aumentará ni disminuirá en función del mercado. Por lo tanto, es útil separar estos conceptos. El precio del gas no está fijado. Los mineros prefieren transacciones con un precio de gas mayor, porque obtendrán mayor recompensa y tardará menos tiempo en confirmase la transacción e incluirse en un bloque.

Al inicio de una ejecución de una transacción, el *transaction_gaslimit* * *gasprice* se restará de la cuenta de usuario. Si la transacción se completa y se ha consumido menos gas del que se ha especificado como límite, el gas restante (*gasrem*) multiplicado por el *gasprice* será devuelto al que envió la transacción.

Y el gas usado realmente (*transaction_gaslimit* - *gasrem*) * *gasprice*, se le dará como premio al minero del bloque.

En el caso de que la transacción supere el *transaction gaslimit*, conocido como "*Runs out of gas*", la transacción se revertirá, y el minero recibirá por el esfuerzo todo el gas (*transaction*_gaslimit*gasprice). El bloque se incluye en la blockchain, pero como fallido.

Adicionalmente, se debe conocer que los propios bloques tienen un campo también de límite de gas llamado ***block gaslimit*** (en Ethereum hablar de *gas limit*, se sobre entiende que se habla del límite de gas de un bloque). Este es el máximo gas combinado por todas las transacciones que permite un bloque aceptar. Este valor actualmente es de 4.712.357 gas, que equivale a un total 224 transacciones que puede tener como máximo un bloque. Los mineros pueden reajustar este valor, aunque lo habitual es que no lo hagan. Si aumentan el *gaslimit*, tardarán más en empezar a minar y si lo eligen más bajo, comenzarán a minar antes (más probabilidades de ganar la carrera) pero cobrarán una menor recompensa.

¿Cómo queda entonces solucionado el problema de los bucles infinitos? Ethereum permite implementar en el código de los *smart contracts* estos bucles. Sin embargo, el atacante con la idea de crear un ataque de DoS, debería tener ether infinito, algo que sería imposible.

Bitcoin, en cambio, la tarifa que se paga por cada transacción es proporcional al tamaño de Kilobytes de la transacción. Cuantas más transacciones contenga un bloque, más se pagará.

Por todo esto, Ethereum es una gran plataforma descentralizada con mucho más potencial que Bitcoin y mucho mejor pensada.

CAPÍTULO 3. DESARROLLANDO SMART CONTRACTS

Entonces, ahora podríamos preguntarnos dónde y cómo programar los *smart contracts*. Cabe recordar, que los *smart contracts* son pequeños programas que se ejecutan en la blockchain y como programa, necesita su lenguaje de programación.

Existen diferentes lenguajes, como lo son **Serpent** (basando en Python), **LLL**, **Bamboo** o **Solidity**. Posiblemente, el más utilizado sea Solidity, un lenguaje de programación de alto nivel creado específicamente por y para Ethereum. Es por este motivo que es el escogido para el desarrollo de los contratos, que posteriormente se convertirán en el núcleo de nuestras aplicaciones descentralizadas.

Pero antes de entrar en profundidad en el desarrollo de *smart contracts*, necesitamos un entorno de trabajo para poder programar y testear los contratos. Y el mejor lugar de trabajo actualmente es **Remix**.

3.1. El entorno de trabajo - Remix

Remix, antes conocido como *Browser Solidity,* es un entorno de desarrollo online basado en navegador con entorno de pruebas y de depuración integrado. Puede instalarse en nuestra propia máquina, pero para más comodidad usaremos la versión online [7].

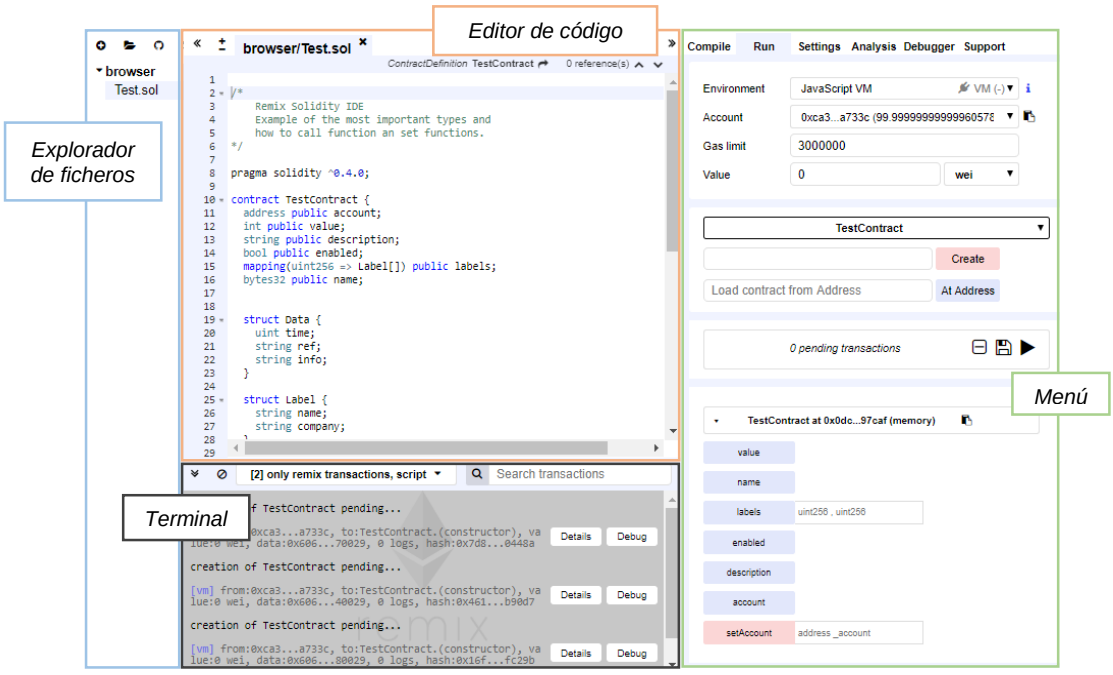

Fig. 3.1 Entorno de desarrollo de Remix (versión final 2017)

Podemos dividir Remix en diferentes partes:

- Explorador de ficheros: Lista por defecto todos los contratos guardados en el navegador. Puedes crear, renombrar y añadir nuevos desde el explorador de archivos.

- Editor de código: Remix recompila el código cada vez que el código sufre un cambio u otro contrato es seleccionado para editarlo. A medida que se va programado, te recompila para avisarte de errores.

- Terminal: Nos muestra las acciones que van sucediendo a medida que se interactúa con Remix y los contratos. Cada vez que se ejecuta un contrato muestra información muy relevante como puede ser: dirección del contrato, dirección del creador, coste de la transacción, coste de la ejecución y *logs*.

- Menú: Remix ofrece diferentes herramientas para poder compilar y ejecutar los contratos. Las dos pestañas más importantes que se han de conocer son las de [*Compile*] y [*Run*].

Desde el Menú de Remix podemos acceder a la pestaña [*Run*] donde se nos mostrará toda la información para interactuar con los *smart contracts* que creemos.

Deberemos ajustar las siguientes opciones:

- ***Environment***: Hemos de decidir en qué entorno de ejecución queremos conectarnos. Tenemos el entorno de **JavaScript VM**, donde en local ejecuta y crea una blockchain de test. **Injected Web3** entorno de ejecución que conecta con algún proveedor que proporciona conectividad con la blockchain de Ethereum como pueden ser *Metamask* o *Mist* (En capítulos posteriores se definirán)**.** Por última opción **Web3 Provider** que permite conectarnos a un nodo remoto.
  El entorno de desarrollo de **JavaScript VM** será el que usemos para los test y el de **Injected Web3** para subir el contrato a la red test de Ethereum.

- ***Account***: Lista de cuentas asociadas el entorno de ejecución que hayamos seleccionado junto con su cantidad de ether asociada

- ***Gas limit***: Máximo gas permitido en las transacciones creadas.

- ***Value*****:** Cantidad de Ether que se quiere usar en una transacción con contrato.

Una vez hemos creamos el contrato, podemos desplegarlo seleccionando el nombre del contrato y pulsando sobre **Create**. Si, por lo contrario, queremos ejecutar un contrato previamente creado, basta con indicar la dirección del contrato y pulsar sobre **At Address**.

Una vez se ejecute el contrato y se suba a la blockchain, nos aparecerán una lista de funciones previamente diseñados en el contrato, para que nosotros podamos interactuar, recuperar datos y añadir datos. Es importante conocer que los botones de color rojo, requieren un consumo de gas por parte de la cuenta que acceda a la función declarada y los de color azul, solo devuelve datos del contrato que se encuentran en la blockchain, por lo tanto, no necesita gastar gas **(Fig.3.1)**.

Hay unas simples reglas que hay que conocer:

- No se pondrán *integers* y *booleanos* entre comillas, excepto si el *integer* es muy grande, se pondrá entre comillas para ser manipulado como *BigNumber*
- Las direcciones tienen que empezar por 0x y tienen que ir entre comillas.
- Las cadenas de caracteres van entre llaves [ ].
- El resto de valores va entre comillas.

El último menú que vamos a utilizar es el de [Compile], que nos será muy útil más adelante. Nos proporciona datos muy importantes del *smart contract* como son algunos metadatos. Los metadatos del contrato incluyen la versión de compilación que se estará usando, El *bytecode* ejecutado durante le creación del contrato, estimaciones de gas y el **ABI** o **ABI array (Aplication Binary Interface)**, el cual describe completamente las interfaces del contrato. Este *array* será muy importante en el desarrollo posterior de las aplicaciones descentralizadas también conocidas como Đapps.

3.2. Solidity

El lenguaje de programación Solidity es muy similar a JavaScript y está orientado a objetos.

La extensión para los ficheros es [**.sol**]. Un contrato de Solidity no es más que una colección de código (funciones y variables) que residen en una dirección especifica de la blockchain de Ethereum. Cada contrato incluirá variables de estado, funciones, modificadores, eventos, estructuras y herencias de otros contratos. Empecemos entendiendo un simple ejemplo básico de almacenamiento de datos en la blockchain para ver el esquema básico de un *smart contract*.

```
// SimpleStorage.sol
pragma solidity ^0.4.0;
contract SimpleStorage {

    uint storedData;

    function set(uint x) {
        storedData = x;
    }
```

```
    function get() constant returns (uint) {
        return storedData;
    }
}
```

La primera línea simplemente dice que el código fuente está escrito para compilar en la versión 0.4.0. El contrato `SimpleStorage` tiene declarada la variable `uint storedData`. Variable que se podrá consultar en la blockchain usando la función `get()` y modificar usando la función `set()`.
La función `get()` devuelve un valor numérico sin modificar ningún valor de la cadena, en cambio, la función `set()`, necesita como parámetro de entrada otro valor numérico para sustituir el valor de la variable.

Este contrato se asimila a una base de datos donde almacenamos un valor y posteriormente lo queremos recuperar. En este caso, una vez publicado este contrato en la blockchain, el valor guardado `storedData` podrá ser consultado por todo el mundo. Por supuesto, cualquier persona puede modificarlo, por lo tanto, no tiene restricciones de acceso impuestas por el propietario del contrato. Más adelante, veremos cómo se pueden imponer restricciones para que solo la persona indicada pueda modificar los datos.

A continuación, se profundizará en los aspectos más relevantes que se necesitan saber para poder crear *smart contracts* a nuestro gusto. En el anexo se puede observar diferentes s*mart contracts* de ejemplo diseñados siguiendo las diferentes características que proporciona Solidity.

3.2.1. Diseño de los ficheros

3.2.1.1. Versión pragma

Todos los contratos empezarán por:

```
pragma solidity ^0.4.0;
```

Especifica que versión del compilador va a ser usada. En este caso, usaremos la versión `0.4.0` o mayor “^” (indicado con el acento circunflejo), pero jamás llegando a usarse la `0.5.0`, para asegurar que no se pierde algunas de las funcionalidades de manera repentina con la nueva versión del compilador.

3.2.1.2. Importar ficheros

- Sintaxis

A nivel global se puede importar de la siguiente manera:

```
import "filename.sol";
```

Esta instrucción importa todas las características definidas en `"filename.sol"` (variables y funciones) al ámbito global actual del contrato.

- Rutas

En lo apartado de sintaxis, el nombre de archivo, si se trata de una ruta, se accede con (`/`) , como separador de directorios, (`.`) para el directorio actual y (`..`) para subir un directorio. Todos los nombres de rutas se tratan como rutas absolutas a menos que comiencen por (`.`) o (`..`) .

Por ejemplo, para importar el fichero `library.sol`, desde el mismo directorio.

```
import "./library.sol";
```

3.2.2. Tipos de datos

Solidity es un lenguaje de tipo estático, lo que quiere decir, que el tipo de variable tendrá que estar definido antes de la compilación. Encontramos los siguientes tipos de datos:

- *Address*

Una `address` ocupa un valor de 20 bytes. Es una variable específica para guardar las direcciones de contratos o de usuarios de Ethereum.

- *Integer*

Un `int` / `uint`, guardan valores numéricos con signo y sin signo respectivamente.

- *String*

El `string` contiene una cadena de caracteres.

- Booleano

Un `bool`, puede tener dos posibles valores (*true* o *false)*.

- *Bytes*

Los `Bytes1` hasta `bytes32`, que son *arrays* de bytes fijos o `bytes` con tamaño variable.

- Enumeración

Los `enum` se pueden usar para crear tipos de datos personalizados con un finito conjunto de valores. La siguiente variable llamada `Estado`, puede tener tres valores.

```
enum Estado { Creado, Modificado, Borrado }
```

- Estructura

`struct` permite construir objetos con múltiples tipos de atributos.

```
struct Objeto {
   bool active;
   uint lastUpdate;
   uint256 debt;
   mapping (address => uint) member;
   . . .
}
```

- *Array* de datos

Los arrays pueden tener un tamaño fijo o pueden ser dinámicos. Una array de tamaño fijo y elemento tipo `T` se escribe como `T[k]`, una matriz de tamaño dinámico como `T[]`.

- Mapeador

El `mapping` básicamente permite usar un tipo de variable como índice de un array. Son declarados como `mapping (KeyType => ValueType)`. Donde `KeyType` es el tipo de dato excepto mapping y el `ValueType` puede ser cualquier tipo.

Por ejemplo, en este caso, La `address` será el índice del array y la entrada de datos será un `integer`.

```
uint tickets;
mapping (address => uint) public purchasers;
. . .
purchasers[0x91e3e0cd9afa1a0ced2456214a2e544a4f5a1db2] += tickets;
```

O también, usando una estructura de datos.

```
struc Purchasers {
   uint timeStamp;
   uint ticket;
}

mapping (address => Purchasers) public purchasers;
```

3.2.2.1. Visibilidad de los datos

Todos los datos pueden tener diferentes tipos de visibilidad para acceder a las variables.

`public`: accesible, se puede obtener el dato guardado en la variable.

`constant`: variable que no se podrá modificar una vez inicializada.

`private`: no accessible.

```
string name;

string public name;
string private name;
string constant name;
```

3.2.3. Unidades y variables especiales disponibles

3.2.3.1. Unidades de Ether

Un número literal puede tomar un sufijo de `wei`, `szabo`, `finney` o `ether` para convertir el número en una subdenominaciones de ether. Sí el número va sin prefijo se asumen wei, la unidad más pequeña.

Algunas conversiones más usadas:

		1000 finey	=	10^3	finey
1 ether	=	1000000 szabo	=	10^6	szabo
		1000000000000000000 wei	=	10^{18}	wei

También se pueden hacer comparaciones entre diferentes unidades, como `2 ether == 2000 finney` y se evalúa como `true`.

3.2.3.2. Unidades temporales

Los sufijos como `seconds`, `minutes`, `hours`, `days`, `weeks` and `years` se pueden usar para convertir numerales en unidades de tiempo, donde los segundos son la unidad base.

3.2.4. Estructura interior de los contratos

El contrato es el objeto más grande que existe en el blockchain de Ethereum. Todas las acciones que permitas hacer con un *smart contract* estarán predefinidas en el `contract`.

Los contratos en Solidity son similares a las clases en lenguajes orientados a objetos. Contienen datos persistentes en variables de estado y funciones que pueden modificar estas variables.

Algunas de las opciones y funciones que nos permitirán diseñar los contratos son las siguientes:

3.2.4.1. Las funciones

Las funciones son las unidades básicas que ejecutarán la parte lógica del código dentro de un contrato. Las funciones que podemos representar vienen creadas siguiendo la regla siguiente:

```
function [functionName] (<parameter types> variable_name)
                  { public | private | internal | external }
                              [ constant | payable | pure | view ]
                                    returns (<return types>)
```

Donde `parameter types`, son los parámetros de entrada que acepta y los `return types`, son los parámetros que devolverá. En ambos casos hay indicar el tipo de variable.

Las funciones pueden ser de diferentes tipos y se especifican como `external`, `public`, `internal` o `private`.

- `public`: Esto le permite definir funciones o variables que se puede llamar internamente o mediante mensajes (transacciones).
- `private`: Las variables y funciones privadas solo están disponibles al contrato actual y no contratos derivados.
- `internal`: Funciones y variables a las que solo se puede acceder internamente (contrato actual o derivado).
- `external`: Funciones que se pueden llamar desde otros contratos y transacciones. No se pueden llamar internamente, excepto con `this.functionName()`.

Otros modificadores de funciones son:

- `payable`: Función que permite recibir ether como parámetro. El ether enviado se almacenará en el contrato. La cantidad de ether almacenada en el contrato se puede ver llamando a `this.balance`.
- `constant`: Define la función únicamente para acceder a datos y no modificarlos. No se malgasta gas, no se mina la transacción.
- `view`: Sustituye a `constant` en la versión 0.4.17 de Solidity.
- `pure`: Para definir funciones donde la lógica de estas funciones no obtiene datos de la blockchain. Siempre que se ejecute esta función con la misma entrada tendrá de datos, obtendremos la misma salida.

Las dos funciones más básicas son:

- Funciones *GET*

Para obtener datos de variables hay dos maneras:

1) Al definir las variables añadirle el nombre la palabra clave public. Pero no es la mejor opción.

```
address public account;
```

   Al ejecutar el contrato automáticamente se creará una función *get* por defecto.

2) Definir una función propia para acceder a los datos de la variable. Se suprimirá la palabra clave public de la variable y en la función se le añadirá la característica de constant.

```
address public account;
function getAccount() constant returns (address){
   return account;
}
```

Una función de tipo `constant` o `view` le indica a la EVM de eliminar la transacción de la blockchain. Esta función de tipo `constant` no producirá ningún cambio, solo devolverá el valor consultado.

Básicamente está devolviendo el valor más reciente conocido, si existe, de la blockchain.
Las funciones *GET* no gastan gas. Si queremos obtener un valor, la EVM creará un bloque virtual donde agrega el último estado, cargará el contrato en la memoria, ejecutará la función y después borrará el bloque virtual. En las funciones GET *n*o se crea ningún bloque.

- Funciones *SET*

Para poder modificar una variable, usamos una función para describir una transacción a la blockchain y salvar permanentemente el valor, por lo tanto modificaran la blockchain. Las funciones SET necesitan parámetros de entrada y no devuelven nada. Las funciones SET gastan gas, como todas las demás funciones.

```
address account;
function setAccount(address _acount) public{
   account = _acount
}
```

3.2.4.2. Métodos de variables y funciones especiales

Hay variables y funciones especiales que siempre existen en el espacio de nombres global y se utilizan principalmente para proporcionar información sobre la blockchain.

- Propiedades de bloques y transacciones

  - `block.blockhash (uint blockNumber) returns (bytes32)`: Hash del bloque dado. Sólo funciona para los 256 bloques más recientes, excluyendo el bloque actual.
  - `block.coinbase (address)`: La dirección del actual del minero del bloque.
  - `block.difficulty (uint)`: Dificultad del bloque.
  - `block.gaslimit (uint)`: Límite de gas actual.
  - `block.number (uint)`: Número del bloque actual.
  - `block.timestamp (uint)`: Marca de tiempo del bloque actual.
  - `msg.gas (uint)`: Gas restante.
  - `msg.sender (address)`: Remitente del mensaje actual.
  - `msg.value (uint)`: Cantidad de Ether (en weis) enviado.
  - `now (uint)`: Alias `block.timestamp (uint)`
  - `tx.gasprice (uint)`: Precio del gas de la transacción.

- Manejo de Errores

  - `assert (bool condition)`: Se lanza si la condición no se cumple.
  - `require (bool condition)`: Se lanza si la condición se cumple.
  - `revert ()`: Cancelar la ejecución y revertir los cambios de estado.

```
modifier onlyOwner {
   if (msg.sender != owner){
      throw;
   else {
      _;
   }
}
```

```
modifier onlyOwner {
   require (msg.sender == owner)
   _;
}
```

El símbolo "_", indica que se aplicar la función donde el modificador está actuando **(ver 3.2.4.4)**.

- Relacionado con las direcciones

  - `balance` y `transfer`:

Es posible consultar el saldo de una dirección usando la propiedad de `balance` y enviar ether (en unidades de wei) a una dirección usando `transfer`.

```
address x = 0x91e3e0cd9afa1a0ced2456214a2e544a4f5a1db2;
address myAddress = this;

if (x.balance < 10 && myAddress.balance >= 10) {
   x.transfer(10);
}
```

  - `send`

El valor `send` indicará si se ha ejecutado o no la transacción. Si la ejecución falla, `send` devolverá *false*.

- `<address>.balance` (uint256): Saldo de la dirección en wei.
- `<address>.transfer` (uint256): Enviar una cantidad dada de wei a la dirección <address>.
- `<address>.send` (uint256 amount) returns (bool): Devuelve *true* si se envía correctamente la cantidad dada.

- Relacionado con el contrato

Métodos que están relacionados con la interacción con los *smart contacts*.

  - `this`: El contrato actual.
  - `selfdestruct (address)` o `suicide (address)`: Destruye el contrato actual enviando los fondos a la dirección dada como parámetro de

entrada. Esto es útil cuando se ha terminado con un contrato, ya que el uso de gas es menor que usando `<address>.send(this.balance)`.

De hecho, el *opcode* SUICIDE utiliza gas negativo porque la operación libera espacio en la cadena de bloques al borrar todos los datos del contrato.

```
address owner;

function kill() {
   selfdestruct(owner);
}
```

Este gas negativo se deducirá del coste total del de la transacción.

3.2.4.3. El constructor

Recordemos que un constructor en programación orientada a objetos. Es una subrutina cuya misión será inicializar un objeto. El constructor ha de tener el mismo nombre que el contrato y será la primera en ser ejecutada. El constructor es opcional y acepta parámetros de entrada. Muy útil para inicializar variables.

```
contract HelloWorld {
   function HelloWorld (type _arg) {
       . . .
   }
}
```

3.2.4.4. Modificadores

Los modificadores se pueden utilizar para cambiar fácilmente el comportamiento de las funciones. Por ejemplo, pueden comprobar automáticamente una condición antes de ejecutar la función.

```
  address public owner;

  function mortal() {
     owner = msg.sender;
  }
  modifier onlyOwner {
     require (msg.sender == owner)
     _;
  }
  function killContract() onlyOwner {
     selfdestruct(owner);
  }
```

En el ejemplo, la función `modifier` ofrece una extensión a la función `killContract()` que solo permite ser ejecutada si se cumple la función `onlyOwner`, es decir, solo se ejecutará por el propietario del contrato. Dentro de la función `modifier` se sustituirá "_" por la función que se quiere ejecutar, en este caso la función `killContract()`.

3.2.4.5. Función Fallback

Esta función no puede tener argumentos y no puede devolver nada. Digamos que es una función por defecto. Se puede usar cuando se quiere enviar una transacción a un contrato de manera sencilla.

```
function() payable { }
```

3.2.4.6. Funciones heredadas

El concepto de herencia en Solidity, significa básicamente que se pueden crear diferentes contratos donde un contrato tiene como herencia las funcionalidades del anterior.

Cuando un contrato hereda de contratos múltiples, sólo se crea un contrato único en la cadena de bloque, y el código de todos los contratos base se copia en el contrato creado.

Se usa el `is` para derivar de otro contrato.

```
contract A {
  address public owner;
  function A() {
     owner = msg.sender;
  }
}

contract B is A {
  function B (type _arg) {
      . . .
  }
}
```

El contrato `B` obtiene las funcionalidades del contrato `A`. `B` adoptará la función definidas en el contrato `A` y las variables globales, como la definida `owner`.

3.2.4.7. Eventos

Desde la interfaz de usuario de la Đapp, podemos registrarnos a un evento que sucede en la blockchain. Dentro de los contratos tenemos sintaxis como lo es `event`, para declarar registros dentro de funciones. Estos registros pueden aceptar parámetros.

```
event Evento (address _address)

function Pay(type _arg) {
    Evento(msg.sender);
}

// Visualización de los eventos desde Remix.Ethereum
Logs
Evento [
  “0x91e3e0cd9afa1a0ced2456214a2e544a4f5a1db2”
]
```

3.2.4.8. Funciones Abstractas

La función abstracta es una función que no tiene implementación real y para heredarla de este contrato es necesario implementarla completamente en el sub contracto o contracto hijo.

```
contract Feline {
    function name() returns (bytes32);
}

contract Cat is Feline {
    function name() returns (bytes32) { return "miaow"; }
}
```

3.2.4.9. Comunicación entre contratos

Una cosa interesante sobre la Ethereum Virtual Machine es que permite crear contratos que se comunican con otros contratos. Esto permite crear aplicaciones descentralizadas más grandes. Se podría tener un contrato que maneje los estados de la aplicación y otro contrato que maneje la lógica y entre ellos se pueden comunicar mediante funciones públicas.

Hay dos maneras de comunicarse ente contratos:

1. Creando una instancia del contrato con el que te quieres comunicar dentro de un nuevo contrato. Esta instancia referenciará el contrato de más abajo.

```
contract Operations {
```

```
    Calculator calc = new Calculator();
    function onePlusSeven() constant returns (int){
        return calc.add(1,7);
    }

contract Calculator {
    function add (int a, int b) returns (int){
       return a + b;
    }
}
```

El contrato *Operations*, obtiene las funciones que permite el contrato *Calculator* y después de instanciar el contrato en la variable `calc`, puede ejecutar las funciones que han sido diseñadas en el contrato *Calculator*.

2. La opción más popular, es crear un contrato y proporcionar la dirección del contrato ya existente en la blockchain.

```
contract Operations {
   Calculator calc = new Calculator();
   Calculator calc = Calculator (0x91e3e0cd9afa1a0ced2456214a2e544…);

   function onePlusSeven() constant returns (int){
      return calc.add(1,7);
}

contract Calculator {
   function add (int a, int b) returns (int){
      return a + b;
   }
}
```

En este caso, primero se implementa el contrato *Calculator* y se incorpora a la blockchain y conociendo su dirección, se incorpora está al contrato *Operations*.

3.2.4.10. Librerías

Con Solidity, es fácil reutilizar código previamente diseñado. Es más eficiente crear una librería para que puedas importarla en el nuevo contrato que se esté desarrollando.

Para crear una librería hay que usar la palabra clave ***library***.

```
library library {
    . . .
}                                                    ./Library.sol
```

Para importar la librería:

```
import ‘./Library.sol’;
contract Example {
    . . .
}                                                    ./Example.sol
```

Este nuevo contrato, ya tendrá definidas todas las funciones y variables de la librería. Se pueden importar librerías desde repositorios de código como Github.

3.3. Desarrollo de los *smart contracts* de Health Hub

La premisa y base de los *smart contracts* que se diseñan para la prueba de concepto de Health Hub es dotar de privacidad a los datos que se incorporarán a blockchain. El diseño funcional es el siguiente:

- Únicamente el creador de los contratos puede incorporar nuevos médicos. El creador del contrato simbólicamente sería la institución médica con capacidad para añadir médicos reales.

- Los médicos jamás podrán ver el historial de un paciente sin consentimiento previo.

- Los doctores podrán añadir nuevos datos al historial clínico del paciente, cuando se les dé acceso.

- Cualquier persona puede inscribirse en el contrato para formar parte de la lista de pacientes.

- El paciente será capaz de añadir médicos para que visualicen sus datos y su historial clínico. Se puede seleccionar, si se desea, que el doctor visualice todo el historial clínico o solo los datos del último año.

- El paciente puede delegar a amigos y familiares, para que, en caso de indisponibilidad, estos den acceso a doctores para consultar los datos.

Para diseñar todas estas funcionalidades se crean 3 *smart contracts* **(Fig.3.2).** Los contratos necesitan obtener información entre ellos. El contrato de los Paciente se nutre de información del contrato de Doctores. Y el contrato de Información, donde se almacena toda la información de los pacientes, se nutre del contrato de Paciente, donde se listan los pacientes existentes.

- **Doctors.sol**: (Contrato para los doctores)

  Se mapean los doctores por su dirección.

  Contiene dos estructuras de datos:
  - Doctor:
    - Nombre del doctor.
    - Lista de pacientes.
    - Dirección del doctor.
    - Especialidad.
  - Paciente
    - Dirección del paciente.
    - Nivel de acceso a los datos.
    - Acceso (True o False).

  Las funciones del contrato son:
  - Añadir doctores.
  - Comprobar si el doctor existe.
  - Buscar información de un doctor.
  - Añadir pacientes a un doctor.
  - Eliminar pacientes de un doctor.
  - Obtener lista de los pacientes de un doctor.

- **Patient.sol**: (Contrato para pacientes)

  Se mapean los pacientes por su dirección.

  Contiene 4 estructuras de datos:
  - Paciente:
    - Nombre del paciente.
    - Edad, Altura y peso.
    - Grupo sanguíneo.
    - Lista de Doctores, Delegados y Delegaciones.
  - Doctor, Delegado y Delegación:
    - Dirección.
    - Nivel de acceso a los datos.
    - Acceso (True o False).

  Las funciones del contrato son:
  - Añadir paciente.
  - Cambiar el peso y la altura.
  - Ver la información del paciente.
  - Añadir un doctor.
  - Añadir y borrar un delegado.
  - Obtener toda la lista de doctores, delegados y delegaciones.
  - Añadir información.

- **Info.sol**: (Contrato para la Información de los pacientes)

  Se mapean lo información junto a la dirección del paciente

Contiene 1 estructuras de datos:

- Información del Paciente:
  - Información añadida.
  - Fecha.
  - Dirección del doctor que añade la información.

Las funciones del contrato son:

- Añadir información al historial clínico.
- Ver el tú propio historial clínico.
- Ver el historial clínico por el paciente.

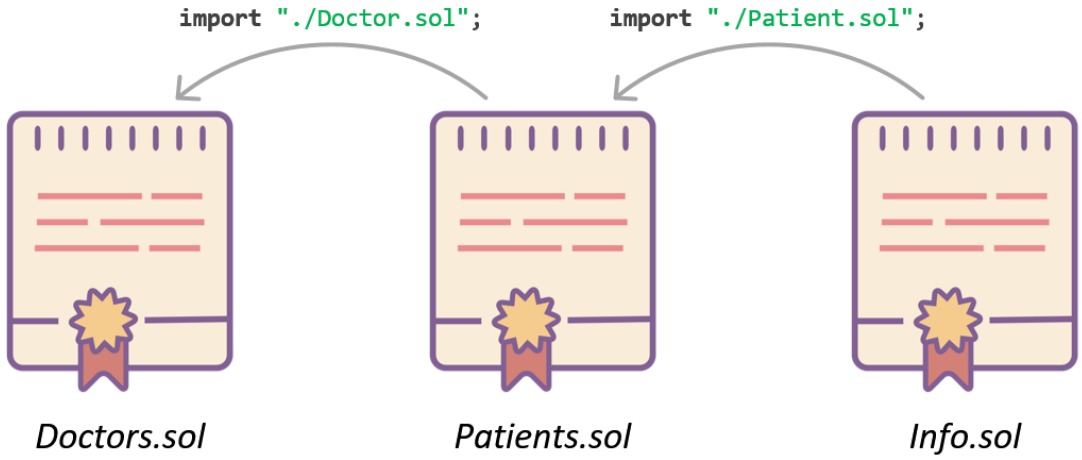

Fig. 3.2 Estructura de los contratos de Health Hub

Los contratos diseñados se pueden encontrar en el anexo del proyecto.

CAPÍTULO 4. APLICACIONES DESCENTRALIZADAS

El siguiente paso a dar, para permitir al usuario final un contacto con la tecnología de una manera más agradable y cercana al uso de blockchain es el de las aplicaciones descentralizadas. Las Đapps (*Descentralized Applications*) son un nuevo paradigma surgido de la tecnología de Ethereum, un nuevo concepto disruptivo.

Primero vale la pena recordar la diferencia entre las aplicaciones web tradicionales y las Đapps. Como todos conocemos, las aplicaciones web o cualquier tipo de aplicación se basan en concepto tradicional de cliente-servidor. La arquitectura básica de aplicaciones o páginas web tradicionales, consta de un portal web o *front-end*, normalmente diseñado en HTML, Javascript y CSS, junto un servidor o *back-end* basado en Java o plataformas como lo son NodeJS **(Fig.4.1)**. El protocolo usado en este caso es el conocido HTTP.

La arquitectura de las aplicaciones descentralizadas, continúa basándose en un *front-end* diseñado en HTML, Javascript, CSS e incorporando una nueva pieza llamada **web3.js**, librería que se encargará de procesar y enviar los datos a la red Ethereum. El protocolo usado en este caso es el llamado RPC. En este caso, el *back-end*, es la blockchain, donde consultar los *smart contracts* de los cuales se nutre una Đapp. Aunque también pueden existir arquitecturas híbridas, donde se hayan de consultar datos no sensibles en servidores externos.

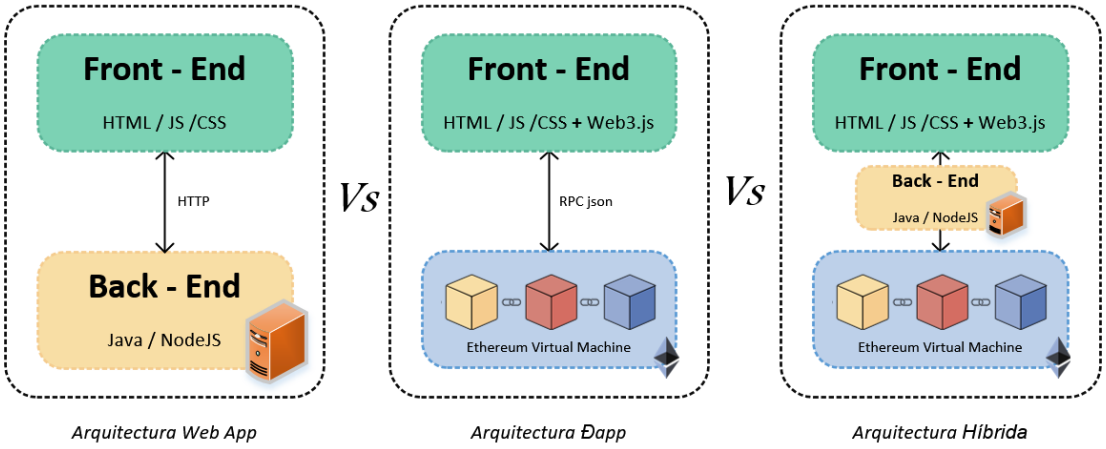

Fig. 4.1 Diferencia entre la arquitectura tradicional y la de una Đapp

Existen miles de aplicaciones que los desarrolladores van creando, la primera de ellas por parte de la propia organización Ethereum fue ***Ethereum Wallet***, que permite hacer transacciones, consultar nuestros fondos y crear e interactuar con los *smart contract*.

Estas aplicaciones, por supuesto siguen necesitando de conexión a Internet y podrán funcionar sin depender de un servidor central, solo nos hará falta disponer de un nodo de Ethereum. Hay diferentes maneras de obtener acceso a la red de Ethereum.

Ethereum nos proporciona la aplicación **Mist** un navegador, para buscar aplicaciones basadas en Ethereum (tanto para Windows como para distribuciones Linux). Mist automáticamente instalará un nodo para interactuar con ellas. Actualmente sigue en desarrollo y se encuentra en la prematura versión 0.9.3. Para versiones basadas en Linux, encontramos herramientas como **Geth**, que nos permite a través de la interfaz de línea de comandos arrancar por completo un nodo de Ethereum.

Otro posible caso sería acceder a un servidor que tenga un nodo de Ethereum y de este modo, conectarnos a él para tener acceso a la blockchain. Recordemos que la blockchain de Ethereum ocupa aproximadamente 100GB a finales de 2017 y posiblemente sea uno de los principales impedimentos a la hora de usar la tecnología, por ello surgió una herramienta para los navegadores tradicionales llamado **Metamask**. Esta extensión que encontramos para Chrome es un puente que interconecta nuestro navegador con Ethereum sin necesidad de descargar ningún nodo. Metamask proporciona una interfaz de usuario para administrar nuestras identidades en diferentes sitios y firmar cada transacción en blockchain.

Ya tenemos todos los conocimientos en la materia, y podemos crear nuestra aplicación descentralizada.

4.1. Desarrollando la Đapp

Health Hub es el nombre que recibirá la aplicación. La premisa es sencilla: nutrir de una interfaz de usuario para que pacientes y doctores puedan compartir historiales clínicos siempre que se cuente con el permiso del paciente o de los delegados que el paciente haya designado.

La herramienta que usaremos para desarrollar las aplicaciones es la conocida como **Meteor**. Meteor es un framework *full-stack*, es decir, con el que se permite programar tanto la capa del servidor (*back-end*) como la capa de la interfaz de usuario (*front-end*). El lenguaje base es Javascript y tiene como objetivo automatizar y simplificar el desarrollo de aplicaciones web. Meteor está integrado con la conocidísima plataforma de gestor de paquetes y dependencias **npm**, pero también tiene el suyo propio llamado **atmosphere**, cuyos gestores usaremos para descargar dependencias extras que necesitaremos para interactuar con la red de Ethereum.

4.1.1. Instalación y configuración

4.1.1.1. Metamask

La extensión de Metamask, actualmente ya está disponible en prácticamente todos los navegadores. Podemos descárgala desde su página oficial *Metamask.io*. Una vez iniciada la extensión debemos introducir una contraseña para crear una nueva cuenta o *wallet*, donde almacenamos el ether. Esta

contraseña será la que nos permita acceder a la aplicación y recuperar el acceso a la cuenta.

Es muy importante guardar la semilla con la que se crea la cuenta, que constará de 12 palabras. Las cuentas en la blockchain se crean de manera aleatoria. No hay manera de ir a un registro y comprobar si esa cuenta ya existe para asignarte una nueva. La probabilidad de que dos cuentas aleatorias se repitieran es de 2^{256}, inimaginablemente grande. Con la semilla que obtendremos podemos crear tantas cuentas como queramos, en nuestro caso podemos necesitar hasta un total de 4 cuentas (propietario del contrato, paciente, paciente delegado y doctor).

Metamask nos permite comunicarnos con todas las blockchain de Ethereum, pero usaremos la de *Ropsten Test Network* para poder ejecutar los contratos tantas veces como queramos sin estar gastando dinero de verdad. A través de la web *Ether Faucet Metamask* podemos enviarnos suficiente ether para desarrollar.

Una vez tengamos en funcionamiento la herramienta, podemos desplegar en la blockchain los 3 contratos diseñados en el apartado **3.3** anterior. Para ello desde la herramienta Remix, indicamos en el campo *Enviroment,* la opción *Injected web3* como vemos indicado en la **(Fig.4.2)**. De esta manera detectará que se está utilizando la extensión Metamask e incorporamos el contrato a la blockchain de test. Un *pop up* aparecerá para que confirmemos la transacción.

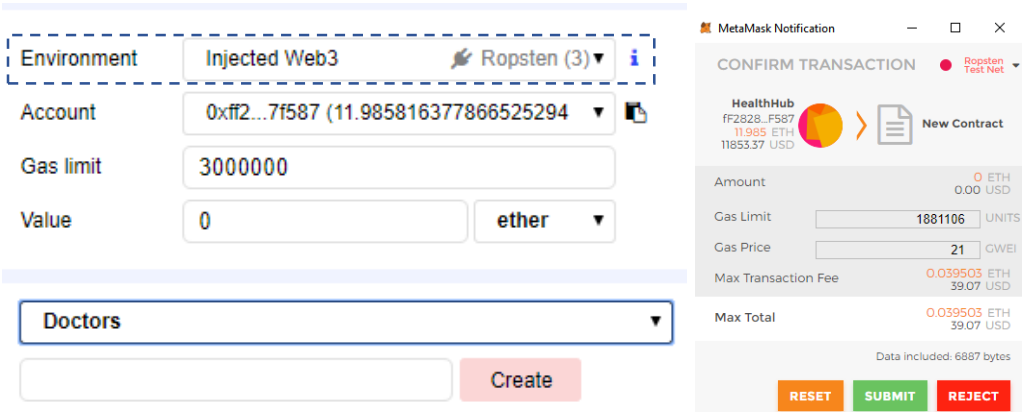

Fig. 4.1 Nueva transacción del *smart contract Doctors.sol*

4.1.1.2. Meteor

Durante el proyecto se ha desarrollado bajo una distribución de Linux, pero por supuesto Meteor es compatible con Windows.

Para instalar la última versión de Meteor desde la línea de comando ejecutaremos lo siguiente:

```
$ curl https://install.meteor.com/ | sh
```

La versión usada durante el desarrollo del proyecto es la 1.6.0.1.

Meteor nos permite crear dos tipos de proyectos, un proyecto más completo con una configuración de carpetas más extendida o un proyecto vacío con las carpetas por defecto para el cliente y el servidor. En este caso nos decantamos por el tipo de proyecto más sencillo.

```
$ meteor create HealthHub
```

A continuación, agregamos las dependencias al proyecto. Por convenio, los nombres de los paquetes de la comunidad incluyen el nombre del mantenedor, junto a dos puntos y el nombre de la dependencia.

- ***ethereum:web3***: Permite conectarnos con nodo de Ethereum y establecer comunicación con los contratos.

Las dos siguientes dependencias, son extras y se usarán para facilitar el desarrollo de la aplicación.

- ***frozeman:template-var*** : Permite mostrar datos en el HTML de manera sencilla sin recargar la página. Derivado de *Handlebars*.

- ***iron:router***: Permite definir rutas para acceder desde el navegador a las diferentes plantillas que diseñemos para la web.

Importante*: Ethereum y las dependencias están en constante desarrollo y puede que muchas de las características dejen de funcionar y se sustituyan por otras en futuras versiones.*

Por ejemplo, para añadir la dependencia de web3.js

```
$ meteor add ethereum:web3
```

La lista de paquetes instalados se encuentra en el fichero `package` dentro de la carpeta `.meteor`.

Por último, para iniciar la aplicación (el puerto por defecto es el 3000, puede definirse añadiendo `--port <Puerto>`).

```
$ meteor run
```

4.1.2. Diseño y estructura

Meteor no fuerza a crear una estructura de carpetas específica, aunque hay algunas carpetas que tienen un significado especial y son tratadas de una manera particular cuando se ejecuta la aplicación.

Por ello hay que conocer la estructura básica y su nomenclatura:

- `/client`: Los ficheros en la carpeta llamada `client` solo serán cargados desde la aplicación del cliente, en este caso desde la web.

- `/lib`: Los archivos que se encuentran dentro de la carpeta `lib`, serán los primeros en cargar en todo el proyecto. Es el lugar idóneo donde cargaremos la librería de Ethereum.

- `/server`: Ficheros que cargarán toda la parte del servidor Meteor, principalmente, el inicio de la propia aplicación.

Una vez conocemos la estructura básica, para el proyecto de Health Hub estructuraremos el proyecto de la siguiente manera:

```
Health Hub
├── .meteor                 # Configuración de Meteor
├── client                  # Colección de archivos de la web
│   ├── css                 # Diseño del estilo de la web
│   │   └── main.css
│   ├── js                  # Lógica detrás de la web
│   │   ├── doctor.js
│   │   ├── patient.js
│   │   └── main.js
│   ├── config.js           # Configuración de los smart contracts
│   ├── doctor.html         # Diseño de la web del doctor
│   ├── patient.html        # Diseño de la web del paciente
│   └── main.js             # Diseño de la portada
├── lib                     # Inicio de la Liberia de Ethereum y
│   ├── init.js               definición de rutas
│   └── routes.js
├── public                  # Imágenes públicas de la web
│   └── images
├── package.json            # Configuración del proyecto
└── ...
```

Fig. 4.2 Estructura de ficheros del proyecto Health Hub

Para el proyecto Health Hub definiremos tres plantillas web, una para el portada de la web, donde el paciente podrá registrarse en la plataforma (`main.js`), una para la página del cliente (`patient.js`) y otra para la del doctor (`doctor.js`). La lógica detrás de cada una de las páginas las encontramos definidas en su correspondiente fichero javascript con el mismo nombre que la plantilla HTML.

En la carpeta `lib`, definiremos las rutas de las aplicaciones para poder acceder a la web (`routes.js`) y cargar la comunicación con el nodo de Ethereum a través de la extensión Metamask (`init.js`).

Por último, en la carpeta `.meteor` encontraremos en el fichero `package`, donde se definen las dependencias del proyecto.

4.1.3. Desarrollo de la Đapp

A continuación, vamos a desarrollar los pasos a seguir para el funcionamiento básico de una Đapp. El código completo de Health Hub es posible localizarlo en el repositorio personal Github[2]

4.1.3.1. Conectar la Đapp con Ethereum

Necesitamos que cada vez que se inicie la web, inicializamos el objeto `web3` con la información del proveedor de Ethereum Metamask mediante la librería web3.js

El fichero `init.js` se encuentra dentro de la carpeta `lib`, este fichero se ejecutará tanto para el cliente como para el servidor, por lo tanto indicamos mediante `Meteor.isClient`, para que solo desde el cliente se inicie el objeto `web3`.

```
if (Meteor.isClient){
  if (typeof web3 !== 'undefined') {
    web3 = new Web3 (web3.currentProvider);
    console.log ("CLIENT CONNECT TO METAMASK");
  } else {
  web3 = new Web3 (new Web3.providers.HttpProvider("http://localhost:8545"));
    console.log ("CLIENT CONNECT TO OWN NODE");
  }
}
```

Si se está usando Metamask, el navegador recupera la información de `currentProvider`. Se comprobará que se ha instanciado el objeto mediante la conexión con Metamask o en el caso que tengamos un nodo instalado, conectándose al puerto 8545 de nuestro ordenador, donde por defecto se inician los nodos de Ethereum.

4.1.3.2. Fichero de configuración

En fichero de configuración `config.js` que encontramos en la carpeta `client`, definiremos 3 objetos donde almacenaremos las direcciones de los 3 contratos que obtenemos desde la propia herramienta Remix una vez añadidos a la blockchain. (Se puede consultar también la dirección de los contratos en la lista de transacciones hechas que nos proporciona Metamask) y junto al contrato añadiremos el ABI array, para poder luego acceder a las funciones definidas en el contrato.

4.1.3.3. Diseño de la web

La web está dividida en tres vistas principales:

[2] Repositorio Health Hub en Github: https://github.com/vmiranda7/health_hub

- **Portada**: *(localhost:3000)*
  Desde la portada principal de la web, se puede acceder automáticamente mediante los dos botones de la parte superior derecha a tanto el menú del paciente o de doctor, según corresponda.
  Mediante el botón situado en la parte central, podemos registrarnos como paciente.

- **Menú del Paciente**: *(localhost:3000/patient)*
  El paciente podrá, mediante el panel izquierdo, ver su información personal, dar acceso a doctores a su historial clínico, agregar delegados para que puedan ofrecer el historial clínico de un paciente a otros doctores y observar que familiares o amigos te han delegado.
  En el panel principal se puede observar detalladamente todo el historial clínico del paciente.

- **Menú del Doctor**: *(localhost:3000/doctor)*
  El doctor podrá ver en su panel izquierdo, su información personal y ver la lista de pacientes a los que tiene acceso para ver su historial médico.
  El historial médico aparecerá en el panel central, según el paciente seleccionado.

Fig.4.3 Diseño de la web Health Hub

Para definir cada una de las vistas de la web, se usan `<templates>`. Son elementos de HTML para mantener el contenido de la web de manera separada, para que posteriormente sea instanciado mediante javascript para renderizar cada uno de las vistas diseñadas según por la ruta web que naveguemos.

Al tener 3 vistas diferenciadas, se proponen 3 `<templates>`.

```
<head>
   <title>Health Hub</title>
   <link rel="shortcut icon" type="images/x-icon" href="images/favicon.ico"/>
</head>

<template name="index">
...
</template>                                                    ./main.html
```

```
<template name="patientInfo">
...
</template>                                                    ./patient.html
```

```
<template name="doctorInfo">
...
</template>                                                    ./doctor.html
```

Los `templates` estarán divididos en los 3 ficheros descritos en el apartado 4.1.1.3. Dentro del código HTML se definen todo el diseño de los elementos de los cuales se compone cada una de las vistas. Para cada una de las vistas se definen de manera separada la lógica detrás en 3 ficheros javascript: `main.js`, `patient.js` y `doctor.js`.

El fichero javascript está divido en 3 funciones principales que ejecuta el `<template>` y se definen como:

```
Template.NombreDelTemplate.[onCreated/helpers/events]
```

Para el caso de la vista definida en el `patient.html`.

```
Template.patientInfo.onCreated(function indexCreated() {
    let contractPatient;
})

Template.patientInfo.helpers({
    patientData(){...}
})

Template.patientInfo.events({
    'click #changeHeight': function(){}
})
```

El método `onCreated()` permite definir todas las funciones y variables que se ejecutarán cuando se visualice el `template`. En el método `helpers()`, se definen las funciones disponibles en la vista. Para invocar a cada una de las funciones se usa la nomenclatura `{{patientData}}` dentro del HTML.

Por último, nos encontramos con el método `events()`, que nos permitirá capturar todas los evento de la web. En este caso, para capturar cuando se pulse sobre un botón, para ejecutar funciones específicas.

4.1.3.4. Definir las rutas

Para definir las rutas usamos la dependencia **iron:router**. Importamos la dependencia en el fichero `routes.js` dentro de la carpeta `lib`.

```
import {Router} from 'meteor/iron:router';

Router.route('<URL>', function(){
  this.render('<Nombre del Template>');
})
```

Definimos 2 rutas para el proyecto (*/patients* y */doctors)*, para acceder a la web de los pacientes y de los doctores **(Fig.4.3)**. Para renderizar la plantilla que se desea, hemos de especificar el nombre del `template` que vamos a usar. La ruta raíz, por defecto nos llevará a la pantalla de inicio.

4.1.3.5. *Interactuar con los smart contracts*

Para interactuar con los *smart contracts* se declararán un objeto por cada contrato incorporado en la blockchain. A través del método `web3.eth` del objeto web3 descrito en el apartado 4.1.3.1 iniciamos una instancia del contrato incorporando la descripción de todas las funciones y eventos que dispone el contrato (ABI array) y la dirección de la blockchain donde está alojado el *smart contract*.

```
contract = web3.eth.contract(<contract.abi>).at(<contract.address>)
```

Una vez definido el objeto `contract`, es posible interactuar con la blockchain de dos maneras:

- Funciones:

Las funciones alojadas en el ABI array pueden ser declaradas en cualquier momento de la siguiente manera: Por ejemplo, si se desea consultar la información de paciente, se define la función en el método `helpers()`.

```
Template.patientInfo.helpers({
    patientData() {
      var name = Template.instance();
       contractInstance.myInfo(function(err, res){
        TemplateVar.set(name, "name", res[0]);
      })
    },
  });
```

Para facilitar la visualización de los datos, usamos la dependencia de **frozeman:template-var**. La ventaja que nos ofrece es directamente modificar los datos de las variables en el HTML de una manera automática. (Solo se permite modificar una variable, no permite arrays).

Primero se inicia el objeto del `template` y posteriormente se añade los datos devueltos por el contrato.

```
TemplateVar.set(name, "name", res[0]);
                #1      #2      #3
```

En el ejemplo anterior, el primer elemento de `TemplateVar.set` es el nombre de la instancia sobre la que se va actuar. El segundo, es el nombre de la variable localizada en el HTML y el último argumento es el valor devuelto por la función `myInfo()`, en este ejemplo en particular.

En el fichero HTML, es necesario llamar a la instancia `TemplateVar` insertando el siguiente código:

```
{{TemplateVar.get "name"}}
```

- Eventos:

También es posible usar los eventos definidos en los contratos, para automáticamente detectar cuando los eventos son lanzados dentro del contrato.

```
var change = contractPatient.MessageChangeData();

change.watch(function(error, result){
  if (!error){
    ...
});
```

El código posterior, define una instancia `change` que, mediante el método `watch`, esperará a activarse en el momento que en la blockchain se lance el evento `MessageChangeData` descrito en el contrato Paciente.

4.1.4. Conclusiones

Con estos simples pasos, es posible diseñar de una manera sencilla aplicaciones descentralizadas. Con esta prueba de concepto basada en privacidad médica se demuestra lo rápido que podemos integrar la tecnología hoy en día. Como el usuario final puede interactuar con los smart contracts y aprovechar todas las características que ofrecen como si una aplicación tradicional se tratará, pero asegurándose que sus datos están descentralizados y no hay una entidad o empresa que almacene los datos y no se conozca que pueden hacer con ellos.

El usuario al final es libre de aceptar y usar las Đapps que necesite y siempre conocerá de qué forma son tratados sus datos realmente. El potencial que tienen los desarrolladores con estas herramientas es infinito.

CAPÍTULO 5. UN FUTURO PROMETEDOR

Desde que se creara Bitcoin en el año 2009, la popularidad de las criptomonedas ha ido en aumento, apareciendo cada vez más proyectos basados en blockchain. Cada vez son más los usuarios que utilizan estas divisas digitales para administrar su capital o llevar a cabo transacciones.

Aunque para muchos el uso de estas monedas virtuales es bastante beneficioso, dentro del marco de la legalidad, aún quedan muchos flecos sueltos, ya que, cuando usamos criptomonedas, lo hacemos dentro de una red descentralizada cuyos movimientos no están regulados por ningún organismo público o entidad financiera.

Sin embargo, el debate de la regulación de las monedas virtuales cada vez está tomando más peso, puesto que, el uso de este tipo de divisas va en aumento y muchos piensan que es necesario crear una normativa para realizar las transacciones, ya que, aunque la utilización de las criptomonedas es muy ventajosa, también a través de ellas se han llevado a cabo acciones delictivas o ilegales.

El 2017 ha sido el gran año de las criptomonedas. Noticiarios, foros y redes sociales, se han llenado de noticias sobre Bitcoin. La sociedad empieza a hablar de Bitcoin, pero no por lo que ofrece la tecnología, sino a causa de la especulación que se está produciendo.

Es probable que Bitcoin se haya convertido en una burbuja. A principios de año, Bitcoin estaba valorado en 1.000$ llegando hasta alcanzar los 19.500$ a mediados de Diciembre [8]. Esto ha sido causado por la masiva capitalización que ha surgido Bitcoin alcanzado los 124 billones de dólares. La sociedad ha empezado a invertir en Bitcoin y ganar grandes cantidades de dinero que ha provocado que más gente en la sociedad, usuarios menos técnicos, también empiecen a invertir como si de acciones se tratarán.

Una semana antes de que Bitcoin tocara su gran cima de casi 20.000$. **Coinbase**, una de las aplicaciones más conocidas que permite conocer el precio de las criptomonedas, hacer transacciones y tanto comprar como vender criptomonedas, supero a la conocida red social *Instagram* en descargas durante una semana [9].

La locura Bitcoin intensificada en los últimos meses ha disparado el interés por la criptomoneda como valor especulativo. Muchos se han interesado en comprar alguna unidad esperando que su valor siga aumentando para poder venderla y recoger las ganancias. Han surgido grandes casas de intercambio de monedes (*exchanges*) que han puesto estas coses más fácil. Con un simple clic, podemos transferir nuestro dinero de la cuenta, intercambiarlo por Bitcoin y recoger las ganancias al cabo de unos meses.

La explosión de Bitcoin ha permitido que muchos desarrolladores hayan empezado a desarrollar nuevas plataformas de blockchain, buscando financiación para su proyecto a través de las conocidas ICOs. En el mundo de las criptomonedas, se utilizar el instrumento conocido como ICO (*Inital Coin Offering*) para financiar el desarrollo de nuevas blockchains o nuevos protocolos. Esto ha permitido también que se alcancen financiaciones de una manera más rápida gracias a que la gente está invirtiendo en criptomonedas, por el mero hecho de ganar dinero y no por los proyectos.

5.1. Blockchains a las que seguir el paso

A pesar de todo esto, no hay que ser negativos. Están apareciendo grandes blockchains que vienen para quedarse, con importantes grupos de desarrolladores que de verdad creen en la tecnología y cabe destacar algunos de ellos.

MONERO

Criptomoneda que nació en el 2014 y que tiene el propósito de aportar de una alternativa centrada absolutamente en la privacidad. Monero es la moneda usada por los cibercriminales. Algunos de los ejemplos vistos este último año, son *botnets* descubiertas que usan los recursos de PCs, portátiles y servidores para minar Monero sin que los usuarios de esas máquinas se den cuenta y sin que se pueda trazar el pago del minado.

NEO

También es conocida como la “Ethereum chino” y se basa también en smart contracts. Neo es la plataforma pionera en China y ha ganado mucha popularidad en Asia. Tiene el respaldo de plataformas como Alibaba o Microsoft. Algunas de las características en comparación a Ethereum es que permite 10 mil transacciones por segundo y Ethereum 15 transacciones de media. La moneda de NEO se llama *GAS*.

ARDOR

Es la primera blockchain escalable. ARDOR es la evolución de NXT, plataforma de test donde desarrollaron esta blockchain. Esta criptomoneda se basa en poder crear subcadenas más pequeñas que se comunican con la cadena padre. De esta manera se reduce el volumen de datos en los nodos y mejora la escalabilidad. La moneda de ARDOR se llama *Ignis*.

ARK

ARK quiere crear un ecosistema de entornos aislados, donde la gente y las startups puedan crear sus propios clones de Blockchain de ARK facilmente con simples clics. Interconectando todo tipo de blockchains mediante puentes inteligentes e implementando los lenguajes de programación más famosos como Solidity. ARK quiere dar

a la gente y a las compañías, todas las herramientas que para expresar sus ideas en el espacio de las criptomonedas y el *Internet of Things*.

5.2. Nueva tecnología más allá del blockchain

Durante el pasado año ha surgido un nuevo sistema que viene a competir con el sistema Blockchain. Este sistema es conocido como **Tangle** y la criptomoneda que la abandera es IOTA.

Tangle también es conocido como DAG (*Directed Acyclic Graph*) es una plataforma de nodos interconectados. *Directed*, significa que las uniones entre bloques siempre tienen una dirección a la que apuntan y *Acyclic* es que no permite bucles cerrados **(Fig.5.1)**.

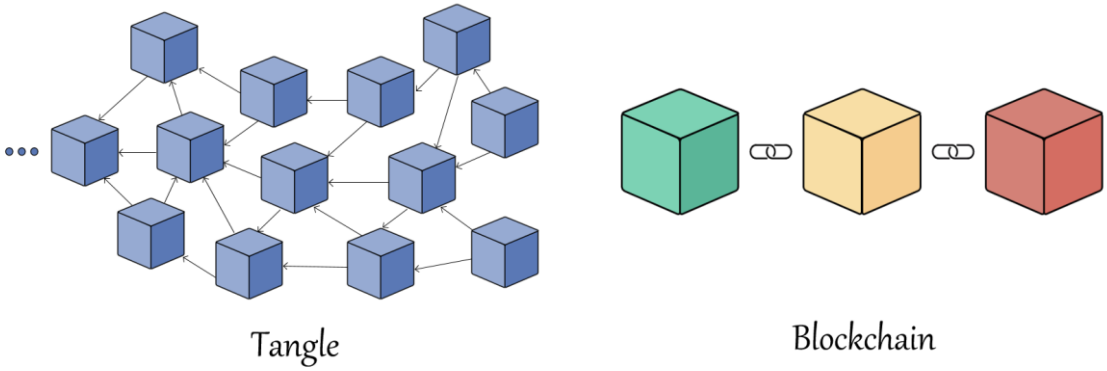

Fig.5.1 Estructura de Tangle y Blockchain

Tangle solventa dos grandes problemas del blockchain:

- **Escalabilidad**: La red de IOTA, es más rápida a medida que van apareciendo más transacciones. Por lo tanto, puede mantener una cantidad de transacciones por segundo ilimitada. Aunque actualmente está alrededor de las 1000.
  Actualmente en blockchain se necesita tener una copia entera de la cadena para empezar a añadir transacciones (esto ARDOR lo pretende solucionar). Actualmente Bitcoin tiene un tamaño de unos 150GB y continuara creciendo. IOTA no necesita una copia entera, solo se necesita una parte para verificar transacciones.

- **Mineria**: IOTA no tiene mineros, para validar los bloques. Por lo tanto no hay una cuota que pagar. Para enviar una transacción de IOTA, el dispositivo de un usuario tan sólo debe confirmar otras dos transacciones en el Tangle. Para confirmar estas dos transacciones, el dispositivo realiza una prueba de trabajo de baja dificultad que es en esencia sólo es serie

de problemas matemáticos. Estos problemas matemáticos pueden ser realizados por casi cualquier dispositivo moderno incluyendo computadoras portátiles y teléfonos.

Tangle aún tiene mucho que demostrar, antes de poder superar a la tecnología Blockchain, pero aun así hay que seguirle la pista.

5.3. El futuro de Ethereum

Definitivamente el 2017 ha sido el año del Bitcoin, pero para el 2018 el Ethereum podría ser la próxima gran criptomoneda y argumentos no le faltan.

Actualmente el capital de mercado de Ethereum es de 70.7 billones de dólares. Y en unos pocos meses posiblemente llegue a superar a Bitcoin. Ethereum tiene también por lo tanto una plan para largo plazo de tres o 5 años según Buterin [10], como el de superar los niveles de escalabilidad de Visa, y en ese caso, desbancar a Bitcoin y convertirse en la principal criptomoneda más importante del mundo.

Pero sin duda, el futuro de esta tecnología quedará en mano de los gobiernos y las legislaciones futuras.

BIBLIOGRAFÍA

[1] Stuart Haber, W. Schoot Stornetta (2009). *How to Time-Stamp a Digital Document* [PDF]. Jounal of Cryptograffy. Disponible en: https://crl.anf.es/pdf/Haber_Stornetta.pdf

[2] Satoshi Nakamoto. (2009). *Bitcoin: A Peer-to-Peer Electronic Cash System* [PDF]. Bitcoin. Disponible en: https://bitcoin.org/bitcoin.pdf

[3] Nick Szabo (1997). *Formalizing and Securing Relations on Public Networks.* First Monday. Disponible en: http://firstmonday.org/ojs/index.php/fm/article/view/548/469

[4] Digiconomist (2018). *Bitocin Energy Consumption Index*. Digiconomist. Disponible en: https://digiconomist.net/bitcoin-energy-consumption

[5] PowerComapare (Noviembre 20, 2017). *Bitcoin Mining Now Consuming More Electricity Than 159 Countries Including Ireland & Most Countries In Africa*. PowerCompare. Disponible en: https://powercompare.co.uk/bitcoin

[6] Dr. Gavin Wood (Enero, 2014). *Ethereum: a secure decentralised generalised transaction ledger* (Apéndice G y H) [PDF]. YellowPaper. Disponible en: http://yellowpaper.io

[7] Ethereum. (s.f). *Remix - Solidity IDE*. Remix-Ethereum. Disponible en: https://remix.ethereum.org

[8] Coinmarketcap (s.f). *Cryptocurrency Market Capitalizations.* Coinmarketcap ©. Disponible en: https://coinmarketcap.com

[9] Kiff Leswing (Diciembre 25, 2017). *It's bitcoin mania: Twice as many people downloaded Coinbase as Instagram last week*. Business Insider. Disponible en: https://amp.businessinsider.com/coinbase-downloads-bitcoin-mania-instagram-2017-12

[10] TrustNodes (Noviembre 25, 2017). *Vitalik Buterin Lays Roadmap for Ethereum Visa Levels Quadratic Sharding.* TrustNodes. Disponible en: https://www.trustnodes.com/2017/11/25/vitalik-buterin-lays-roadmap-ethereum-visa-levels-quadratic-sharding

[11] Fundación CTIC (Enero 25, 2017). *¿Qué es el "blockchain" del que todo el mundo habla?* CTIC. Disponible en: http://www.fundacionctic.org/ctic/articulos-y-otras-publicaciones/que-es-el-blockchain-del-que-todo-el-mundo-habla

[12] Directivos y Gerentes (Mayo 18, 2017). *Todo lo que tienes que saber sobre Blockchain, la última gran revolución tecnológica*. dir&ge. Disponible en: https://directivosygerentes.es/digital/articulos-digital/lo-tienes-saber-blockchain-la-ultima-gran-revolucion-tecnologica

[13] Ethereum (s.f). *White Paper*. Github. Disponible en: https://github.com/ethereum/wiki/wiki/White-Paper

[14] Criptonoticias (s.f). *¿Qué es Ethereum?* Criptonoticias. Disponible en: https://criptonoticias.com/informacion/que-es-ethereum

[15] Wikipedia (Setiembre 23, 2017). *Prueba de trabajo (algoritmo de consenso distribuido)*. Wikipedia, La enciclopedia libre. Disponible en: https://es.wikipedia.org/wiki/Prueba_de_trabajo_(algoritmo_de_consenso_distribuido)

[16] Ethereum (s.f). Dapp using Meteor. Github. Disponible en: https://github.com/ethereum/wiki/wiki/Dapp-using-Meteor

[17] Ethereum (s.f). JavaScript API. Github. Disponible en: https://github.com/ethereum/wiki/wiki/JavaScript-API

[18] Dr. Gavin Wood (s.f) *Ethereum: a secure decentralised generalised transaction ledger* [PDF]. Gavwood. Disponible en: http://gavwood.com/paper.pdf

[19] Adrián Arroyo (Marzo 2, 2016). *Ethereum y SmartContracts*. El Blog de Adríán Arroyo. http://adrianistan.eu/blog/2016/03/02/ethereum-smart-contracts-contratos-inteligentes

[20] Elena Cazes (Mayo 30, 2017). *Metamask. El puente entre Ethereum y tu navegador.* Criptonoticias. Disponible en: https://www.criptonoticias.com/aplicaciones/metamask-puente-ethereum-navegador

[21] Jérôme Jehrli (Noviembre 23, 2016). *Blockchain 2.0, From Bitcoin Transactions to Smart Contracts*. SlideShare. Disponible en : https://es.slideshare.net/JrmeKehrli/blockchain-20-69472625

[22] Bit2Me (s.f). *Smart Contracts. Una guía para principiantes*. Bit2Me. Disponible en: http://blog.bit2me.com/es/smart-contracts

[23] Carlos González Juárez (Agosto 19, 2017). *Solidity language for Ethereum*. Medium. Disponible en: https://medium.com/@Carlos_molotov/solidity-language-for-ethereum-c434cf03ea8c

[24] Ethereum (s.f). *web3js*. Github. Disponible en: https://github.com/ethereum/web3.js/

[25] Ethereum (s.f). *Mist*. Github. Disponible en: https://github.com/ethereum/mist

[26] Ethereum (s.f). *Ethereum Blockchain app platform*. Ethereum Organization. Disponible en: https://ethereum.org

[27] Ethereum (s.f). *Ethereum Homested Documentation.* Ethereum Docs. Disponible en: http://ethdocs.org

[28] Ethereum (s.f). *Remix*. Remix Ethereum. Disponible en: https://remix.ethereum.org

[29] Ethereum (s.f). *Browser-solidity*. Github. Disponible en: https://github.com/ethereum/browser-solidity

[30] Yann300 (2017). *Remix – Solidity IDE*. Remix.ReadTheDocs. Disponible en: http://remix.readthedocs.io/en/latest/run_tab.html

[31] My Ether Wallet (s.f). *What is Gas?* MyEtherWallet. Disponible en: https://myetherwallet.github.io/knowledge-base/gas/what-is-gas-ethereum.html

[32] Ethereum (s.f). *Solidity.* Solidity.ReadTheDocs. Disponible en: http://solidity.readthedocs.io/en/latest/index.html

[33] Antonio Madeira (Enero 12, 2018). *What is the "gas" in Ethereum?* CryptoCompare. Disponible en: https://www.cryptocompare.com/coins/guides/what-is-the-gas-in-ethereum

[34] Hudson Jameson (Junio 27, 2017). *Accounts, Transactions, Gas and BLockc Gas Limits in Ethereum*. Husdonjameson.com. Disponible en: https://hudsonjameson.com/2017-06-27-accounts-transactions-gas-ethereum

[35] Afri (Noviembre 29, 2017). *The Ethereum-blockchain size will exceed 1TB anytime soon.* Dev. Disponible en: https://dev.to/5chdn/the-ethereum-blockchain-size-will-not-exceed-1tb-anytime-soon-58a

[36] Ameer Rosic (Junio 2017). *What is an Ethereum token: the ultimate Beginner's guide.* Blockgeeks. Disponible en: https://blockgeeks.com/guides/what-is-blockchain-technology

[37] Nolan Baurele (s.f) *What is the Blockchain Techonology?* Coindesk. Disponible en: https://www.coindesk.com/information/what-is-blockchain-technology

[38] Savjee (Noviembre 20, 2017). *Smart contracts – Simply Explained*. YouTube. Disponible en: https://www.youtube.com/watch?v=ZE2HxTmxfrI

[39] Savjee (Diciembre 26, 2017). *IOTA's Tangle – Simply Explained*. YouTube. Disponible en: https://www.youtube.com/watch?v=CZxH1V_zoug

[40] Cryptografi (Agosto 7, 2017). *Solidity in 2 Minutes.* YouTube. Disponible en: https://www.youtube.com/watch?v=3i203iTmcFc

[41] Mobilefish.com (Agosto 24, 2017). *Ethereum gas, gas limit, gas price*. YouTube. Disponible en: https://www.youtube.com/watch?v=yFb2nuUUDX0

[42] Blockchain at Berkelley (Junio 18, 2017). *Deep Dive: Ethereum and Smart Contracts*. YouTube. Disponible en: https://www.youtube.com/watch?v=2jisWLxf38E

[43] Ethereum Foundation (Noviembre 26, 2017). *Intro to Solidity 2017 Edition*. YouTube. Disponible en: https://www.youtube.com/watch?v=KkN1O8TChbM

[44] Tapscott. D y Tapscoot. A. *La revolución del blockchain*. Deusto, Barcelona (2017)

ANEXOS

1. Contratos de Ejemplo

Ejemplos de *smart contracts* diseñados usando las propiedades descritas en el apartado 3 del proyecto.

1.1. Hello World

Smart contract básico que nos da la posibilidad de guardar en la blockchain un valor, consultarlo y modificarlo por cualquier usuario.

```
pragma solidity ^0.4.0; // 0.4.0 o mayor pero no la 0.5.0

contract HelloWorld {

  string name;

  // Constructor que por defecto asigna a [name] el nombre de "World"
  function  HelloWorld () {
    name = "World";
  }

  // Función GET
  // constant = indicar a la EVM de eliminar la transacción de la blockchain
  // porque esta función no está cambiando nada, solo está devolviendo el valor.
  function sayHello() constant returns (string, string) {
    return ("Hello", name);
  }

  // Función SET
  // Modifica la variable [name] por la introducida por el usuario.
  function setName (string n) {
    name = n;
  }
}
```

1.2. My Hello World

Smart contract básico que nos da la posibilidad de guardar en la blockchain un valor, consultarlo, pero esta vez sólo puede modificarlo el usuario que creo el contrato, que es el propietario del contrato.

```
pragma solidity ^0.4.0; // 0.4.0 o mayor pero no la 0.5.0

contract HelloWorld {

  string word = "Hello World";
  address issuer; // Editor del contrato

  // Constructor que por defecto al iniciar el contrato guarda como
  // [issuer] el creador del contrato (su dirección pública)
  function MyHelloWorld() {
```

```
    issuer = msg.sender;
  }
```

```
  // Modificador que concede permisos únicamente para el creador del contrato
  modifier ifIssuer(){
    require (issuer == msg.sender);
    _;
  }

  // Función GET
  // Devuelve el valor [word]
  function getWord() constant returns (string) {
    return word;
  }

  // Función SET
  // Modifica el valor [word] si y solo si eres el creador del contrato
  function setWord (string w) ifIssuer returns (string) {
    word = w;
    return "Correct Modified";

  }
}
```

1.3. Depósito de ether

El siguiente *smart contract* permite depositar fondos por cualquier usuario y retirarlos y visualizar la cantidad de fondos, solo por el usuario que creo el contrato. Es como un ejemplo de *crowdfunding* básico.

```
pragma solidity ^0.4.0; // 0.4.0 o mayor pero no la 0.5.0

contract CustodialContract {

  address client;

  // Registros donde se almacenan mensajes que pueden ser usados en interfaces de Đapps
  event Message (string msg);
  event UpdateStatus(string _msg, uint amount);
  event UserStatus(string _msg, address user, uint amount);

  // Constructor que por defecto al iniciar el contrato guarda como
  // [client] el creador del contrato (su dirección pública)
  function CustodialContract() {
    client = msg.sender;
  }

  // Modificador que concede permisos únicamente para el creador del contrato
  modifier ifClient() {
    require(msg.sender == client);
     _;
  }

  // Función para depositar Ether. Para que se puedan depositar fondos es necesarios
  // indicar que es una función del tipo payable.
  function depositFunds() payable {
        UserStatus("User transferred some money", msg.sender, msg.value);
  }

  // Función para recuperar fondos guardados en el contrato.
```

```
  // El usuario no puede recuperar más fondos de los que hay en el contrato guardado.
  function witdrawFunds(uint amount) ifClient {
    if (client.send(amount)) {
      UpdateStatus("You witdraw", amount);
    }
    else {
      UpdateStatus("You can not witdraw", amount);
    }
  }

  // Función para visualizar los fondos del contrato (this.balance).
  // Los fondos solo puede visualizarse por el creador del contrato, gracias al
  // modificadorde ifClient
  function getFunds() ifClient constant returns(uint) {
    Message("You see the funds");
    return this.balance;
  }
}
```

1.4. Interacción entre contratos

La posibilidad de crear un contrato nuevo con funciones que se comunican con otro contrato y lo ejecutan. En este caso tenemos un contrato que se genera primero *CalledContract* y cuando tenemos ese contrato generado, el contrato *CallerContract* lo generamos a partir de la dirección pública del primer contrato.

```
pragma solidity ^0.4.0; // 0.4.0 o mayor pero no la 0.5.0

//Primer contrato a generar
contract CalledContract {

  uint number = 7;
  bytes32 word = "Seven";

  // Devuelve el valor de [number]
  function getNumber() constant returns (uint) {
    return number;
  }

  // Devuelve el valor de [word]
  function getWords() constant returns (bytes32) {
    return word;
  }
}

// Segundo contrato a generar con la dirección pública del contrato CalledContract
contract CallerContract {

  // Instancia donde se guardara el contrato generado anteriormente
  CalledContract toBeCalled;

  // Constructor
  // Guardar la instancia del contrato CalledContract en función de la dirección publica
  function CallerContract(address _addressContract){
    toBeCalled = CalledContract(_addressContract);
  }

  // Devuelve el valor de [number] llamando al contrato CalledContract
  function getNumber() constant returns(uint){
    return toBeCalled.getNumber().toStri;
  }
```

```
  // Devuelve el valor de [word] llamando al contrato CalledContract
  function getWords() constant returns (bytes32){
    return toBeCalled.getWords();
  }
}
```

1.5. El Concierto

Smart contract de un Concierto con la posibilidad de comprar entradas, devolverlas y consultar la dirección de la página web del concierto. Cuando se vendan todas las entradas, el creador del contrato se llevara toda la recaudación obtenida y el *smart contract* se cancelará para quedar inservible.

```
pragma solidity ^0.4.0; // 0.4.0 o mayor pero no la 0.5.0

contract Concert {

  address owner;
  uint public tickets;
  uint constant price = 1 ether; // 1*10**18 wei
  // Address es el indicador único y del número de tickets comprados
  mapping (address => uint) public purchasers;

  // Constructor
  // Hay que indicar cuantos tickets estarán a la venta al ejecutar el contrato
  function Concert (uint t) {
      owner = msg.sender;   // Propietario del concierto que recibirá todo el dinero.
      tickets = t;          // Total de tickets disponibles
  }

  // Se pueden definir funciones sin nombre
  // La función principal es permitir enviar dinero al contrato directamente sin
  // ejecutar ninguna función, pero que este hecho de enviar ether nos haga comprar una
  // entrada.
  function() payable {
      buyTickets(1);
  }

  // Función de comprar tantas entradas como indiquemos
  function buyTickets(uint amount) payable {
    // Comprueba si el número de entradas compradas y la cantidad que pagas es la
    // correcta
    // Comprueba también si hay entradas disponibles
    require(msg.value == (amount * price) && amount <= tickets);
    // Guarda quien ha comprador las entradas y la cantidad
    purchasers[msg.sender] += amount;
    // Disminuye la cantidad de entradas disponibles
    tickets -=amount;
    if (tickets == 0){
        // Si se han vendido todas las entradas, se destruye el contrato
        // Por defecto el Ether acumulado en el contrato es de vuelta al propietario
        selfdestruct(owner);
    }
  }

  // La función abstracta es una función que no tiene una implementación real y para
  // heredarla de este contrato debe implementarla en su totalidad en AbstarctContracts
  function website() returns (string);

}


// Funciones definidas sin implementación que pueden ser heredadas por otros contratos
```

```
interface Refundable {
  function refund (uint numTickets) returns (bool);
}

// Contrato hereda del contrato Concierto
// Hereda todas las funciones y variables y esta inicializada con 10 entradas
// Implementa Refundable que es la función que hemos definido como interfaz
contract AbastracContract is Concert(10), Refundable {

    // Definiendo la funcion heredada de la interfaz Refundable
    // Permite devolver los Ether de las entradas
    function refund(uint numTickets) returns (bool) {
      // Si devuelve más de los que había comprado = False
      if (purchasers[msg.sender] < numTickets){
          return false;
      }
      // Si devuelve menos o los que había comprado = True
      msg.sender.transfer(numTickets*price);
      purchasers[msg.sender] -= numTickets;
      // Se volverán a aumentar el número de entradas
      tickets += numTickets;
      return true;
    }

    // Función heredada de Concert que devuelve la web del Concierto
    function website() returns (string) {
        return "https://www.theconcert.com";
    }
}
```

1.6. El Banco

Smart contract que crea un banco para que los usuarios depositen su dinero. Vean el balance de ether y retiren el dinero.

```
pragma solidity ^0.4.0;

contract SimpleBank {

  //
  mapping (address => uint) private balances;
  address[] allAddress;
  uint _balance;
  address public owner; // Externamente leíble pero no editable


  // Registros donde se almacenan mensajes que pueden ser usados en interfaces de Đapps
  event LogDepositMade(address accountAddress, uint amount);
  event NewAddress(string msgs, address newAddres);
  event Send(uint total, uint withdraw);

  // Constructor
  function SimpleBank() {
      owner = msg.sender; // Propietario del concierto que recibirá todo el dinero.
  }

  // Modificador que concede permisos únicamente para el creador del contrato
  modifier onlyOwner {
    require(msg.sender == owner);
    _;
  }
```

```
  // Función depositar dinero
  function deposit() payable public returns (uint) {
    if (msg.value > 0 ether) {
      bool search = false;
      for (uint i=0; i<allAddress.length; i++) {
        if (allAddress[i] == msg.sender){
          search = true;
        }
      }
      if (!search) {
        allAddress.push(msg.sender);
        NewAddress("NewAddress", msg.sender);
        _balance += msg.value;
      }
      balances[msg.sender] += msg.value;
      LogDepositMade(msg.sender, msg.value);
      return balances[msg.sender];
    }
  }

  // Función recuperar dinero del Banco
  function withdraw(uint withdrawAmount) public returns (uint remainingBal) {
    if (balances[msg.sender] >= withdrawAmount) {
      _balance -= withdrawAmount;
      Send(balances[msg.sender],withdrawAmount);
      msg.sender.transfer(withdrawAmount);
      balances[msg.sender] -= withdrawAmount;
      return balances[msg.sender];
    }
    else {
      return balances[msg.sender];
    }
  }

  // Función para saber que usuarios disponen de dinero en el banco
  function addresses() constant returns (address[]) {
    return allAddress;
  }

  // Función para saber cuánto dinero depositado en el banco se tiene
  function balance() constant returns (uint) {
    return balances[msg.sender];
  }

  // Función para obtener el dinero del que dispone el Banco
  // Solo visible por el propio Banco (Creador del contrato)
  function BanckBalance() onlyOwner returns (uint) {
    return _balance;
  }
}
```

2. Contratos de Health Hub

A continuación se muestran los 3 contratos creados para la prueba de concepto de Health Hub desarrollada.

2.1. Contrato de los Doctores (Doctor.sol)

```
pragma solidity ^0.4.0;

contract Doctors {

    struct Doctor {
        string name;
        string speciality;
        address doctorAddress;
        mapping (address => Patient) mypatient;
        uint IDlength;
        Patient[] patient;
    }

    mapping(address => Doctor) public doctors;

    struct Patient {
        uint ID;
        address patientAddress;
        uint level;
        bool access;
    }

    event Message(string msgs);

    address owner;

    function Doctors(){
        owner = msg.sender;
    }

    modifier onlyOwner() {
      require(msg.sender == owner);
      _;
    }

    function addDoctor(string _name, string _speciality, address _doctorAddress)
        onlyOwner   {
        doctors[_doctorAddress].name = _name;
        doctors[_doctorAddress].speciality = _speciality;
        doctors[_doctorAddress].doctorAddress = _doctorAddress;
        Message("Added Correctly");
    }

    function myInfo() constant returns (string, string, address)  {
        return ( doctors[msg.sender].name,
                 doctors[msg.sender].speciality,
                 doctors[msg.sender].doctorAddress);
    }

    function existDoctor (address _doctoraddress) constant returns(uint)  {
        if (doctors[_doctoraddress].doctorAddress == 0){
            return (1); // NO EXIST
        }
        else{
            return (0); // EXIST
        }
    }

    function addPatient(address _patientAddress, address _doctorAddress, uint _level)
        external {

        if(doctors[_doctorAddress].mypatient[_patientAddress].access != true){
```

```
            uint id = doctors[_doctorAddress].IDlength;

            doctors[_doctorAddress].mypatient[_patientAddress].patientAddress =
              _patientAddress;
            doctors[_doctorAddress].mypatient[_patientAddress].level = _level;
            doctors[_doctorAddress].mypatient[_patientAddress].access = true;
            doctors[_doctorAddress].mypatient[_patientAddress].ID = id;

            Patient memory newPatient = Patient({
              ID: id,
              patientAddress: _patientAddress
              level: _level,
              access: true
          });

            doctors[_doctorAddress].patient.push(newPatient);
            doctors[_doctorAddress].IDlength += 1;
            Message("PatientAdded");
        }

    }

    function delPatient(address _patientAddress, address _doctorAddress) external {

    uint idP = doctors[_doctorAddress].mypatient[_patientAddress].ID;

    if(doctors[_doctorAddress].patient[idP].patientAddress == _patientAddress
       && doctors[_doctorAddress].patient[idP].access == true){

            doctors[_doctorAddress].patient[idP].access=false;
            delete doctors[_doctorAddress].mypatient[_patientAddress];
            Message("PatientDeleted");
            return;
        }

    }

    function getAllPatient() constant returns (address[], uint[]) {
        address[] patientList ;
        uint[] patientListLevel ;

        patientList.length = 0;
        patientListLevel.length = 0;

        for (uint i=0;i<doctors[msg.sender].patient.length; i++ ){
            if(doctors[msg.sender].patient[i].access == true){
            patientList.push(doctors[msg.sender].patient[i].patientAddress);
            patientListLevel.push(doctors[msg.sender].patient[i].level);
            }
        }
        return (patientList, patientListLevel);
   }

}
```

2.2. Contrato de los Pacientes (Patient.sol)

```
pragma solidity ^0.4.0;

import "./Doctor.sol";

contract Patients {
```

```
    struct Patient {
        address patientAddress;
        string name;
        uint age;
        uint height;
        uint weight;
        string bloodType;

        Info[] info;
        mapping (address => uint) mydoctor;
        Doctor[] doctor;
        mapping (address => uint) mydelegate;
        Delegate[] delegate;
        mapping (address => uint) mydelegated;
        Delegated[] delegated;
    }

    struct Info {
        string info;
        uint date;
        address doctorAddress;
    }

    struct Doctor {
        address doctorAddress;
        uint level;
        bool access;
    }

    struct Delegate {
        address delegateAddress;
        uint level;
        bool access;
    }

    struct Delegated {
        address delegatedAddress;
        uint level;
        bool access;
    }

    mapping(address => Patient) private patients;

    event MessageExistDoctor(string msgs);
    event MessageError(string msgs);
    event Message(string msgs);
    event MessageChangeData(uint msgs);

    Doctors doc;

    function Patients(address _doctorContract)  {
        doc = Doctors(_doctorContract);
    }

    /////////////////////////// PACIENTE ////////////////////////////

    function addPeople(string _name, uint _age, uint _height, uint _weight,
        string _bloodType) {
        patients[msg.sender].patientAddress = msg.sender;
        patients[msg.sender].name = _name;
        patients[msg.sender].age = _age;
        patients[msg.sender].height = _height;
        patients[msg.sender].weight = _weight;
        patients[msg.sender].bloodType = _bloodType;
        Message("Added Correctly");
    }
```

```
    function changeHeight(uint _height) {
        patients[msg.sender].height = _height;
        MessageChangeData(_height);
    }

    function changeWeight(uint _weight)   {
        patients[msg.sender].weight = _weight;
        MessageChangeData(_weight);
    }


    function myInfo() constant returns (address, string, uint, uint, uint, string){
        return (patients[msg.sender].patientAddress,
        patients[msg.sender].name,
        patients[msg.sender].age,
        patients[msg.sender].height,
        patients[msg.sender].weight,
        patients[msg.sender].bloodType);
    }

    // COMPROBAR
    function myInfoToDoctor (address _patientAddress) constant returns (address,
        uint, uint, uint, string){
        if (patients[_patientAddress].mydoctor[msg.sender] != 0 ){
            return (patients[_patientAddress].patientAddress,
                patients[_patientAddress].age,
                patients[_patientAddress].height,
                patients[_patientAddress].weight,
                patients[_patientAddress].bloodType);
        }
    }

//////////////////////////// (ADD/DEL/LIST) DOCTOR ////////////////////////////

    function addDoctor(address _doctorAddress, uint _level) {
        if(doc.existDoctor(_doctorAddress) == 0
           && patients[msg.sender].mydoctor[_doctorAddress] == 0){
            if (_level == 1 || _level == 2){
                patients[msg.sender].mydoctor[_doctorAddress] = _level;

                Doctor memory newDoctor = Doctor({
                  doctorAddress : _doctorAddress ,
                  level: _level,
                  access: true
                });
                patients[msg.sender].doctor.push(newDoctor);

                doc.addPatient(msg.sender, _doctorAddress, _level);
                Message("DoctorAdded");
            }else{
                MessageError("Error-01");
                return;
            }
        }
        else{
            MessageExistDoctor("Doctor does not exist");
        }
    }

    //EL PACIENTE BORRA UN DOCTOR
    function delDoctor(address _doctorAddress) {
        for (uint i=0; i<patients[msg.sender].doctor.length; i++){

            if(patients[msg.sender].doctor[i].doctorAddress == _doctorAddress
                && patients[msg.sender].doctor[i].access == true ){
```

```
                patients[msg.sender].doctor[i].access=false;
                delete patients[msg.sender].mydoctor[_doctorAddress];
                doc.delPatient(msg.sender,_doctorAddress);
                Message("DoctorDeleted");
                return;
            }
        }
    }

    function getAllDoctor() constant returns (address[], uint[]) {
        address[] doctorList ;
        uint[] doctorListLevel ;

        doctorList.length = 0;
        doctorListLevel.length = 0;

        for (uint i=0;i<patients[msg.sender].doctor.length; i++ ){
            if(patients[msg.sender].doctor[i].access == true){
            doctorList.push(patients[msg.sender].doctor[i].doctorAddress);
            doctorListLevel.push(patients[msg.sender].doctor[i].level);
            }
        }

        return (doctorList, doctorListLevel);
    }

    ////////////////// (ADD/DEL/LIST) DELEGATE AND DELEGATED /////////////////


    function addDelegate(address _delegateAddress, uint _level) {
        if (patients[msg.sender].mydelegate[_delegateAddress] == 0){

            patients[msg.sender].mydelegate[_delegateAddress] = _level;
            patients[_delegateAddress].mydelegated[msg.sender] = _level;

            Delegate memory newDelegate = Delegate({
                delegateAddress : _delegateAddress ,
                level: _level,
                access: true
            });

            Delegated memory newDelegated = Delegated({
                delegatedAddress: msg.sender,
                level: _level,
                access: true
             });

            patients[msg.sender].delegate.push(newDelegate);
            patients[_delegateAddress].delegated.push(newDelegated);
            Message("DelegateAdded");

        }else{
            MessageError("NotAdded");
            return;
        }
    }

    function delDelegate(address _delegateAddress) {
        for (uint i=0; i<patients[msg.sender].delegate.length; i++){
            if(patients[msg.sender].delegate[i].delegateAddress == _delegateAddress
                && patients[msg.sender].delegate[i].access == true ){
                patients[msg.sender].delegate[i].access=false;
                delete patients[msg.sender].mydelegate[_delegateAddress];
                delDelegated(_delegateAddress, msg.sender);
                Message("DelegateDeleted");
```

```
                return;
            }
        }
    }

    function delDelegated(address _delegateAddress, address _patientAddress) {
      for (uint i=0; i<patients[_delegateAddress].delegated.length; i++){
       if(patients[_delegateAddress].delegated[i].delegatedAddress == _patientAddress
          && patients[_delegateAddress].delegated[i].access == true ){
                patients[_delegateAddress].delegated[i].access=false;
                delete patients[_delegateAddress].mydelegated[_patientAddress];
                Message("DelegatedDeleted");
                return;
            }
        }
    }

    function getAllDelegate() constant returns (address[], uint[]) {
        address[] delegateList ;
        uint[] delegateListLevel ;

        delegateList.length = 0;
        delegateListLevel.length = 0;

        for (uint i=0;i<patients[msg.sender].delegate.length; i++ ){
            if(patients[msg.sender].delegate[i].access == true){
            delegateList.push(patients[msg.sender].delegate[i].delegateAddress);
            delegateListLevel.push(patients[msg.sender].delegate[i].level);
            }
        }
        return (delegateList, delegateListLevel);
    }

    function getAllDelegated() constant returns (address[], uint[]) {
        address[] delegatedList ;
        uint[] delegatedListLevel ;

        delegatedList.length = 0;
        delegatedListLevel.length = 0;

        for (uint i=0;i<patients[msg.sender].delegated.length; i++ ){
            if(patients[msg.sender].delegated[i].access == true){
            delegatedList.push(patients[msg.sender].delegated[i].delegatedAddress);
            delegatedListLevel.push(patients[msg.sender].delegated[i].level);
            }
        }
        return (delegatedList, delegatedListLevel);
    }


    function addDoctorBeingDelegate(address _doctorAddress, address _patientAddress){
        if(doc.existDoctor(_doctorAddress) == 0){
            if(patients[_patientAddress].mydelegate[msg.sender]!= 0){
                uint _level = patients[_patientAddress].mydelegate[msg.sender];
                patients[_patientAddress].mydoctor[_doctorAddress] = _level;

                Doctor memory newDoctor = Doctor({
                   doctorAddress : _doctorAddress ,
                   level: _level,
                   access: true
                });
                patients[_patientAddress].doctor.push(newDoctor);

                doc.addPatient(_patientAddress, _doctorAddress, _level);
                Message("DoctorAdded");
```

```
            }
        }
        else{
            Message("Doctor does not exist");
        }
    }

   ////////////////////////////// INFO //////////////////////////////////

   function addInfo (address _patientAddress, address _doctorAddress) external
      returns (uint){
        for(uint i=0; i < patients[_patientAddress].doctor.length;i++){
            if(patients[_patientAddress].doctor[i].doctorAddress == _doctorAddress
                && patients[_patientAddress].doctor[i].access == true){
                return(0);
            }
        }
    }

  function isDoctor(address _patientAddress, address _doctorAddress) external
      constant returns(uint){
        if(patients[_patientAddress].mydoctor[_doctorAddress] != 0){
            if(patients[_patientAddress].mydoctor[_doctorAddress] == 1){
                return(1);
            }
            else{
                return(2);
            }
        }
    }
}
```

2.3. Contrato de la información de Pacientes (Info.sol)

```
pragma solidity ^0.4.0;

import "./Patient.sol";

contract InfoPatient {

    struct InfoP {
        Info[] info;
    }
    struct Info {
        string info;
        uint date;
        address doctorAddress;
    }

    mapping(address => InfoP) infopatient;

    Patients pat;

    event MessageH(string msg);

    function InfoPatient(address _patientContract){
        pat = Patients(_patientContract);
    }

    function addInfo (address _patientAddress, string _info){
        if(pat.addInfo(_patientAddress, msg.sender) == 0){
            Info memory newInfo = Info({
              info : _info ,
```

```
            date: now,
            doctorAddress: msg.sender
        });
        infopatient[_patientAddress].info.push(newInfo);
        MessageH("InfoAdded");
        return;
    }
}

function getMyInfoLength() constant returns (uint){
    return(infopatient[msg.sender].info.length);
}

function getMyInfo(uint i) constant returns(string, uint, address){
    return (infopatient[msg.sender].info[i].info,
    infopatient[msg.sender].info[i].date,
    infopatient[msg.sender].info[i].doctorAddress);
}

function getMyInfotoDoctorLength(address _patientAddress) constant returns(uint){
    if(pat.isDoctor(_patientAddress, msg.sender) == 1){
        return(infopatient[_patientAddress].info.length);
    }
    else{
        uint consultDate = now;
        uint ldata = infopatient[_patientAddress].info.length;
        uint oneyear = 365 * 24 * 60 * 6 * 10;
        uint infoL;
        uint diff = consultDate- oneyear;
        for(uint k=ldata; k>0; k--){
            if(infopatient[_patientAddress].info[k-1].date > diff){
                infoL++;
            }
            else{
                return(infoL);
            }
        }
    }
}

function getMyInfotoDoctor(address _patientAddress, uint i) constant
    returns(string, uint, address){
        if(pat.isDoctor(_patientAddress, msg.sender) == 2){
            uint consultDate = now;
            uint tenMinuts = 365 * 24 * 60 * 6 * 10;
            uint diff = consultDate-oneyear;
            if(infopatient[_patientAddress].info[i].date > diff){
                return (infopatient[_patientAddress].info[i].info,
                    infopatient[_patientAddress].info[i].date,
                    infopatient[_patientAddress].info[i].doctorAddress);
            }
        }
        else{
            return (infopatient[_patientAddress].info[i].info,
                    infopatient[_patientAddress].info[i].date,
                    infopatient[_patientAddress].info[i].doctorAddress);
        }
}

}
```