Universidad Zaragoza

1542

Trabajo Fin de Grado

# Desarrollo de Smart Contracts en una Blockchain basados en información semántica

# Development of Smart Contracts for a Blockchain based on semantic information

Autor

Álvaro Fuentemilla Martínez

Directores

Fernando Bobillo Ortega

ESCUELA DE INGENIERÍA Y ARQUITECTURA
2019

# RESUMEN

Blockchain es una tecnología relativamente nueva de la que aún hay muchas aplicaciones por descubrir. Uno de los sectores en los que está más asentada en el sector financiero, desde la llegada de Bitcoin en 2009 son muchas las criptomonedas que han ido apareciendo, siendo Libra, creada por Facebook, una de las más recientes.

Sin embargo, los usos que se le pueden dar a esta tecnología son múltiples, básicamente, cualquier tipo de información que necesite ser preservada de forma intacta y que deba permanecer disponible puede ser almacenada en una Blockchain de manera segura y descentralizada. Con la creación de Ethereum en 2015, empieza a ser posible la creación de Smart Contracts ya que esta Blockchain integra un lenguaje Turing completo. La mayoría de estos contratos se programan en Solidity que está orientado específicamente a contratos que se ejecutan en la Máquina Virtual de Ethereum(EVM).

Con este trabajo se quiere conseguir cambiar la implementación de los Smart Contracts para poder desarrollarlos con información semántica. Este tipo de implementación facilitaría su comprensión y creación. Para ello será necesario crear una ontología capaz de representar las partes que componen un Smart Contract, utilizar la Blockchain Ethereum para las transacciones llevadas a cabo en los contratos y finalmente almacenar el contrato para que no pueda ser modificado utilizando la tecnología IPFS.

Además de conseguir implementar un Smart Contract clásico en el que una transacción se realiza solo cuando se cumplen unos términos previamente acordados, se ha conseguido crear acuerdos parciales mediante el uso de lógica difusa.

# Índice

# Capítulo 1

# Introducción y objetivos

## 1.1. Motivación

Blockchain es es una base de datos distribuida que registra bloques de información y los entrelaza para facilitar la recuperación de la información y la verificación de que ésta no ha sido cambiada. La cadena de bloques es almacenada por todos aquellos nodos de la red que se mantienen en sincronía con ésta. Se trata de un término que en pocos años ha pasado de ser desconocido a resultar familiar. Son muchas las empresas y gobiernos que están invirtiendo en esta tecnología para investigar todas sus posibles utilidades, que debido a las características que representa Blockchain descentralizado, seguro y anónimo, son muy diversas. Desde servicios financieros, sanidad, música, propiedad intelectual, sistemas electorales, automoción e Internet de las Cosas (IoT), entre otros.

El principal uso que se le ha dado a esta tecnología es el financiero a través de criptomonedas, siendo Bitcoin la más conocida. En 2008, se creo el Bitcoin y a partir de ese momento fue donde se empezó a conocer el Blockchain e investigar todo su potencial. Desde la aparición de esta criptomoneda han ido surgiendo otras y otro tipo de Blockchain como es el caso de Ethereum.

Ethereum es una Blockchain que no solo permite hacer transacciones sino que además permite la creación de Smart Contracts, contratos inteligentes, o aplicaciones distribuidas donde los datos están distribuidos por la red. Los Smart Contracts son contratos que se almacenan en la Blockchain con el fin de que ninguna de las dos partes lo modifique, actualmente la mayoría de estos contratos se programan en un lenguaje llamado Solidity.

## 1.2. Objetivos

El principal objetivo de este trabajo es el estudio de la gestión de información semántica y la interacción con razonadores semánticos a la hora de implementar Smart

Contracts. Se trata de encontrar maneras alternativas a Solidity para programar dichos contratos. A través de la información semántica se conseguirá definir de forma clara y concisa las partes que intervienen en un contrato, facilitando así la implementación del mismo.

Otro punto a tener en cuenta es la aplicación de lógica difusa para llegar a acuerdos parciales. En un Smart Contract normal si se cumplen unas condiciones previamente estipuladas se suceden unos hechos que normalmente suelen ser transacciones entre las dos partes afectadas, utilizando la lógica difusa se podría conseguir que no se tuvieran que cumplir sí o sí unas condiciones sino que se llegara un acuerdo que satisficiera a las dos partes.

Para obtener estos objetivos, se siguió este proceso: después de una fase previa de documentación acerca de Blockchain se decidió que la mejor para trabajar con Smart Contracts era Ethereum. Por lo tanto se pasó al estudio de esta cadena de bloques y su funcionamiento. Primero, hubo que entender bien el concepto de Smart Contracts y ver qué requisitos había que cumplir. Ver las posibilidades que ofrece Solidity y cómo se podrían replicar utilizando ontologías. Para solventar el problema del almacenamiento del Smart Contract se decidió que la forma más eficiente era utilizar la tecnología IPFS ya que al tratarse de archivos grandes no es fácil almacenarlos directamente en la Blockchain. Posteriormente se tuvo que encontrar la forma de interactuar con la Blockchain y la ontología a través de Java. Una vez conseguida la creación de un Smart Contract clásico se pasó a la implementación de un Smart Contract con acuerdos parciales a través de FuzzyDL.

## 1.3. Trabajo relacionado

Los estudios realizados actualmente que utilicen conjuntamente Blockchain e información semántica no tienen como objetivo la realización de Smart Contrats con información semántica. Algunos de estos trabajos realizados tienen los siguientes propósitos:

- Utilizar la información semántica para describir la estructura nativa de Blockchain y la información relacionada. [11]

- Generar Smart Contracts de forma autómatica con la ayuda de ontologías y reglas semánticas, en este caso la información semántica sirve de apoyo a la hora de realizar el Smart Contract. A través de la ontología se establecen las restricciones que tiene el contrato, siendo estas restricciones a menudo reutilizadas en diferentes contratos de un área de aplicación determinada. Sin embargo, con

esta aplicación de la información semántica no se consigue que sea la ontología sea la que represente el Smart Copntract. [10]

- Crear un índice que proporciona la capacidad de búsqueda en la Blockchain. [13]

## 1.4. Descripción del documento

El documento se encuentra estructurado de la siguiente forma. En los capítulos 3, 4, 5 y 6 la estructura es similar, se empieza con una descripción de la tecnología que se va a usar y posteriormente se explica como se ha utilizado y adaptado para alcanzar los objetivos.

- El Capítulo 2 se centra en la técnología Blockchain y su funcionamiento.
- El Capítulo 3 habla de la red Ethereum y cómo se va utilizar para la realización de los Smart Contracts.
- El Capítulo 4 explica la ontología utilizada y describe cada una de las partes que la conforma.
- El Capítulo 5 desarrolla la aplicación de la lógica difusa a los Smart Contract.
- El Capítulo 6 trata del almacenamiento del contrato en IPFS.
- Conclusiones obtenidas del trabajo realizado.
- En el anexo se incluye un diagrama de Gantt.

# Capítulo 2

# Blockchain

Blockchain, como su propio nombre indica, es una cadena de bloques que sirve para almacenar y transmitir datos en la red de una forma distribuida y segura. [17]

A diferencia de una red centralizada en la que todo depende del nodo central, en una red distribuida todos los nodos crean una malla interconectada, dando redundancia en el servicio. Los diversos nodos de la red replican la información de la cadena de bloques consiguiendo así un consenso sobre la integridad de los datos por parte de todos los participantes de la red sin necesidad de recurrir a un sistema de confianza que centralice la información.

Cada bloque que forma la cadena (excepto el bloque generatriz, que inicia la cadena) contiene un código hash correspondiente al bloque de la cadena anterior en una línea temporal, una cantidad de registros o transacciones válidas, información referente a ese bloque y otro código hash que enlazará con el siguiente bloque, un código único que sería como la huella digital del bloque. Cada bloque tiene un lugar específico e inamovible dentro de la cadena, ya que cada bloque contiene información del hash del bloque anterior. La información contenida en un bloque solo puede ser repudiada o editada modificando todos los bloques anteriores.

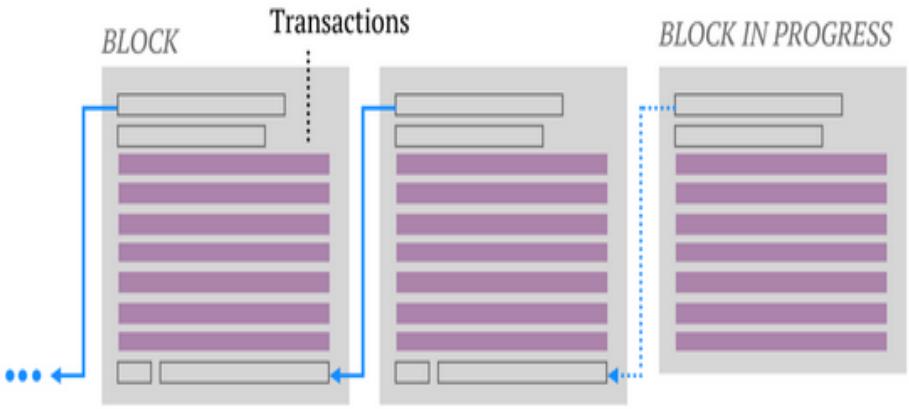


Figura 2.1: Cadena de bloques

Todos los nodos que forman parte de la red Bitcoin mantienen una lista colectiva de todas las transacciones conocidas en una cadena de bloques. Para ello los nodos generadores de los bloques, también llamados mineros, meten en la cabecera de dichos bloques el hash o resumen del último bloque de la cadena más larga de la que tienen conocimiento, así como las nuevas transacciones publicadas en la red. Cuando un minero encuentra un nuevo bloque, lo transmite al resto de los nodos a los que está conectado. En el caso de que resulte un bloque válido, estos nodos lo agregan a la cadena y lo vuelven a retransmitir. Este proceso se repite hasta que el bloque ha alcanzado todos los nodos de la red. Por tanto, la cadena de bloques contiene el historial de posesión de todas las monedas desde la dirección-creadora a la dirección del actual dueño. Por lo tanto, si un usuario intenta reutilizar monedas que ya usó, la red rechazará la transacción.

El objetivo concreto de los mineros es buscar un *nonce* (es un campo de 32 bits cuyo valor se establece de modo que el hash del bloque contenga una ristra de ceros) correcto para el bloque de forma que el bloque completo satisfaga cierta condición. Esta condición para el caso de Bitcoin es que el doble hash SHA-256 del bloque tenga un cierto número de ceros iniciales. Este proceso de prueba-error hace difícil la generación de bloques. Si consiguen encontrar este hash reciben una recompensa en forma de criptomoneda; en el caso del bitcoin, por ejemplo, cada vez que un minero encuentra un bloque válido es recompensado con 12,5 bitcoins. El pago se realiza con monedas que se encuentran en reserva y en ese momento entran en circulación, es por esto que erróneamente se suele creer que la minería de criptomonedas consiste en generar nuevas monedas. Las monedas ya están previamente definidas, sin embargo, a través de la minería se logra que entren en circulación nuevas monedas. [17]

Actualmente la mayoría de Blockchains que utilizan esta tecnología como una herramienta financiera, es decir, una criptomoneda como es el caso de la más conocida: Bitcoin. Sin embargo, existen Blockchains que se crearon con otros objetivos, Ethereum permite la creación de acuerdos de contratos inteligentes entre pares, gracias a que integra un lenguaje Turing completo.

Los usos que se le pueden dar a esta tecnología son múltiples, básicamente, cualquier tipo de información que necesite ser preservada de forma intacta y que deba permanecer disponible puede ser almacenada en una Blockchain de manera segura, descentralizada y más económica que a través de intermediarios. Algunas de las propuestas que se están investigando son las siguientes: [19]

- Seguridad social y salud: los registros de salud podrían ser unificados y almacenados en una Blockchain. De esta forma, la historia médica de cada

paciente estaría segura y a la vez disponible para cada médico autorizado, independientemente del centro de salud donde se haya atendido el paciente. Incluso la industria farmacéutica puede utilizar esta tecnología para verificar medicamentos y evitar falsificaciones.

- Registro de propiedades: el gobierno japonés ha iniciado un proyecto para unificar todo el registro de propiedades urbanas y rústicas con tecnología de cadena de bloques, lo que permitiría contar con una base de datos abierta en la que se pudieran consultar los datos de las 230 millones de fincas y 50 millones de edificios que se estima existen en el país asiático. En Dubai están planeando algo muy parecido. [3] [2]

- Contratación pública: la DGA quiere introducir esta tecnología en el marco de un contrato público: se solicitaría a los licitadores la presentación de la huella electrónica (hash) de su oferta, quedando almacenada de manera permanente, simultánea y sucesiva en un sistema distribuido en varios nodos (Blockchain). [4]

# Capítulo 3

# Ethereum

En las secciones 3.1, 3.2 y 3.3 de este capítulo se describe la Blockchain Ethereum así como los Smart Contracts actuales programados en Solidity. En las secciones 3.4 y 3.5 se explica el trabajo realizado para poder trabajar con la Blockchain.

## 3.1. Ethereum

Ethereum es una blockchain pública y descentralizada que permite la creación de acuerdos de contratos inteligentes entre pares. En esta plataforma cualquier desarrollador puede crear y publicar aplicaciones distribuidas que realicen contratos inteligentes. Ethereum funciona de manera descentralizada a través de una máquina virtual llamada Ethereum Virtual Machine (EVM). Esta máquina ejecuta un código intermedio o bytecode el cual es una mezcla de LISP, ensamblador y bitcoin script. Los programas que realizan contratos inteligentes se escriben en lenguajes de programación de alto nivel de tipo Turing completos, como Serpent o Solidity, que siguen la metodología de diseño por contrato para crear los contratos inteligentes. Los Smarts Contracts, son compilados a dicho bytecode y subidos a la red de Ethereum, donde pueden ser ejecutados por cualquier persona. [5]

Ethereum usa como divisa interna el Ether, la criptomoneda descentralizada subyacente al mismo que sirve para ejecutar los contratos del mismo. A este respecto, Ethereum no es como la mayoría de las criptodivisas existentes, ya que no es solamente una red para reflejar las transacciones de valor monetario, sino que es una red para la alimentación de los contratos basados en Ethereum. Estos contratos de código abierto pueden ser usados para ejecutar de forma segura una amplia variedad de servicios, entre los que se incluyen: sistemas de votación, intercambios financieros, plataformas de micromecenazgo, propiedad intelectual y organizaciones descentralizadas autónomas. En Ethereum, los desarrolladores también pueden escribir la lógica de negocio y acuerdos en forma de contratos inteligentes, los cuales se ejecutan automáticamente

cuando sus condiciones son satisfechas por ambas partes; posteriormente dichas condiciones se informan a la red. Estos contratos pueden almacenar datos, enviar y recibir transacciones e incluso interactuar con otros contratos, independientemente de cualquier control.

## 3.2. Smart Contract

Hasta ahora los contratos han sido documentos verbales o posiblemente caros documentos escritos, sujetos a las leyes y jurisdicciones territoriales, y en ocasiones requiriendo de notarios, es decir, más costes y tiempo. Algo no accesible para cualquier persona. En algunos casos los contenidos de los contratos pueden estar sujetos a la interpretación.

En cambio, un contrato inteligente es un programa informático capaz de ejecutarse y hacerse cumplir por sí mismo, de manera autónoma y automática, sin intermediarios ni mediadores y que ejecuta acuerdos establecidos entre dos o más partes haciendo que ciertas acciones sucedan como resultado de que se cumplan una serie de condiciones específicas. Es decir, cuando se da una condición programada con anterioridad, el contrato inteligente ejecuta automáticamente la cláusula correspondiente. Evitan el lastre de la interpretación al no ser verbal o escrito en los lenguajes que hablamos. Los smart contracts se tratan de scripts escritos con lenguajes de programación, siendo los términos del contrato puras sentencias y comandos en el código que lo forma.

Estos smart contracts viven en una atmósfera no controlada por ninguna de las partes implicadas en el contrato, en un sistema descentralizado. Esto significa que:

1. Se programan las condiciones.
2. Se firman por ambas partes implicadas.
3. Y se almacena en una blockchain para que no pueda modificarse.

Y por otra parte, tienen como objetivo principal:

- Implementar una estado de seguridad mayor al del contrato tradicional.
- Reducir costes.
- Reducir el tiempo asociado a este tipo de interacciones.

## 3.3. Solidity

Solidity es un lenguaje de programación de alto nivel orientado a contratos. Su sintaxis es similar a la de JavaScript y está enfocado específicamente a la Máquina

Virtual de Etehreum (EVM), se compila en código de bytes (bytecode) que es ejecutado en la EVM. [12] Solidity está tipado de manera estática y acepta, entre otras cosas, herencias, librerías y tipos complejos definidos por el usuario. Mediante Solidity, los desarrolladores pueden escribir aplicaciones descentralizadas que implementen automatizaciones en los negocios a través de los contratos inteligentes, dejando un registro irrefutable y autorizado de las transacciones.

## 3.4. Rinkeby

Existen varias redes de prueba para Ethereum, se decidió instalar Rinkeby porque era la más fácil para conseguir Ether de prueba y da soporte a Geth.

Para su instalación se siguieron los siguientes pasos:

1. Instalar Geth, que es una interfaz de línea de comandos (CLI) que sirve como cliente para ejecutar un nodo completo de Ethereum implementado en Go.

   Usaremos Go para:

   - Correr un nodo Ethereum conectado a la red de prueba Rinkeby.
   - Crear cuentas capaces de enviar y recibir transacciones.
   - Para leer el estado de la EVM, por ejemplo, comprobar el balance de una cuenta.

   Para la instalación de geth se utilizó el siguiente comando:

```
$ brew tap ethereum/ethereumbrew install ethereum
```

2. Inicializar el nodo que se va a usar para las pruebas:

```
$ geth --rinkeby --datadir=~/.gophersland_ethereum_r1
--port=30304 --cache=2048 --rpc --rpcport=8546
--rpcapi=eth,web3,net,personal --syncmode=light
```

   El comando anterior:

   - Inicializa un nuevo directorio donde se guardarán los datos $\sim/.gophersland\_ethereum\_r1$. El directorio por defecto es $\sim/ethereum$.
   - Empieza a descargar el historial de Ethereum necesario para convertirse en un nodo válido y sincronizado de la red.
   - La comunicación ocurre a través del puerto 30304.

- La memoria caché se configura a 2GB para acelerar el proceso de sincronización.
- Se lanzará una API RPC adicional para que podamos comunicarnos con nuestro nodo a través de consolas, GUI en el puerto 8546.
- El modo de sincronización es syncmode=light, es el que obtiene solo el estado actual. Para verificar los elementos, debe solicitar a los nodos completos las hojas de árbol correspondientes. Descarga solo el encabezado de los bloques y no descarga todos los bloques. Para todo tipo de procesamiento, este tipo de nodo depende de los pares de nodo completo. Por su parte tiene la ventaja del poco tiempo que tarda en sincronizarse en torno a 10 min, en contraposición de las horas que puede tardar otro modo de sincronización como el "fast", y el poco espacio de almacenamiento que utiliza, 300 MB.

3. Abrir la GUI para poder comunicarse con el nodo a través del siguiente comando:

```
$ geth attach ~/.gophersland_ethereum_r1/geth.ipc
```

4. Una vez abierta la GUI hay que comprobar que el nodo esta sincronizado: si el siguiente comando devuelve false significa que ya está sincronizado, en caso contrario devolverá el número de bloque en el que se encuentra la sincronización:

```
$ eth.syncing
```

5. Crear una nueva cuenta y comprobar el balance de la misma. Para crear una nueva cuenta se utilizará el siguiente comando, que una vez ejecutado pedirá una contraseña para la nueva cuenta:

```
$ geth account new
```

Una vez creada se podrá ver su balance accediendo a la GUI de la siguiente forma:

```
$ eth.getBalance("0xE6534A87481064250c4983853aAbFb9d9d4B53f2")
```

El número hexadecimal que se le pasa al comando corresponde con la dirección de la cuenta creada previamente. Un error frecuente que puede ocurrir es que si no se encuentran suficientes pares que sean nodos completos, la petición de ver el balance fallará debido a que el modo de sincronización light solo obtiene el estado actual como se ha comentado anteriormente.

6. En caso de que el apartado anterior haya acabado en error habrá que añadir los pares necesarios:

```
$ admin.addPeer("direccion del nodo completo");
```

```
$ admin.peers
```

Con el segundo comando se observa si realmente se han añadido los pares.

Una vez se han completado estos pasos el nodo esta listo para poder trabajar con él.

Para poder llevar a cabo transacciones entre las diferentes cuentas creadas es necesario obtener Ether que poder transferir. Para conseguir Ether gratuito en la página de Rinkeby, https://www.rinkeby.io, en la opción de Crypto Faucet compartiendo en una red social la dirección de la cuenta a la que quieres que se te transfiera el Ether cada cierto tiempo, (3 Ethers / 8 horas, 7.5 Ethers / 1 día o 18.75 Ethers / 3 días) se puede obtener Ether para la red de prueba de forma gratuita.

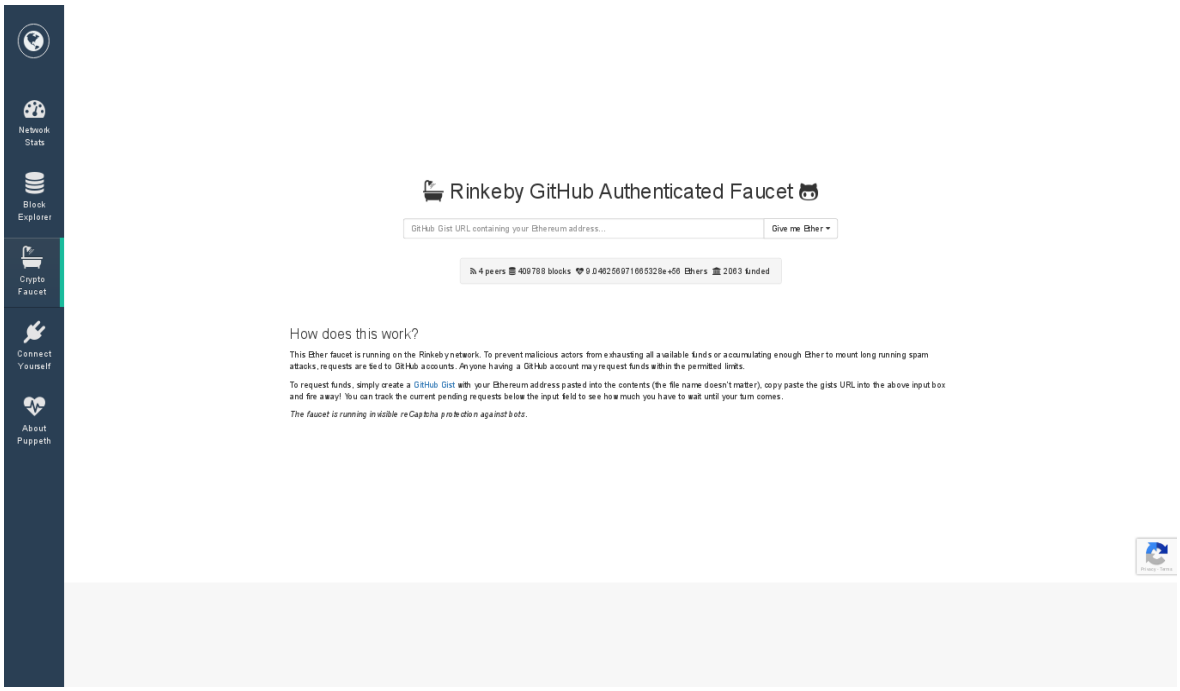


Figura 3.1: Rinkeby Faucet

## 3.5. Web3j

Para poder interactuar a través de Java con la red Ethereum se ha utilizado la siguiente biblioteca, web3j [14]. Web3j cuenta con las principales funciones funciones necesarias para poder llevara a cabo un Smart Contract:

- Conectarse a un nodo previamente inicializado.
- Crear una nueva cuenta.
- Comprobar el balance de una cuenta.
- Realizar transferencias entre dos cuentas.

Al principio también se iba a usar esta biblioteca para comunicarse con Smart Contracts creados en Solidity ya que no se conocía la posibilidad de transferir Ether, entonces la idea era crear una función en Solidity que realizará la transferencia de Ether, finalmente se pudo realizar la transferencia a través de Java.

Para poder usar la biblioteca a través de Eclipse, se han seguido las indicaciones del GitHub para el uso de la biblioteca y se ha creado un proyecto Maven y añadido las siguientes dependencias:

```
<dependency>
  <groupId>org.web3j</groupId>
  <artifactId>core</artifactId>
  <version>4.2.0</version>
</dependency>
```

En la Figura 3.2 se observa la estructura de la clase implementada:

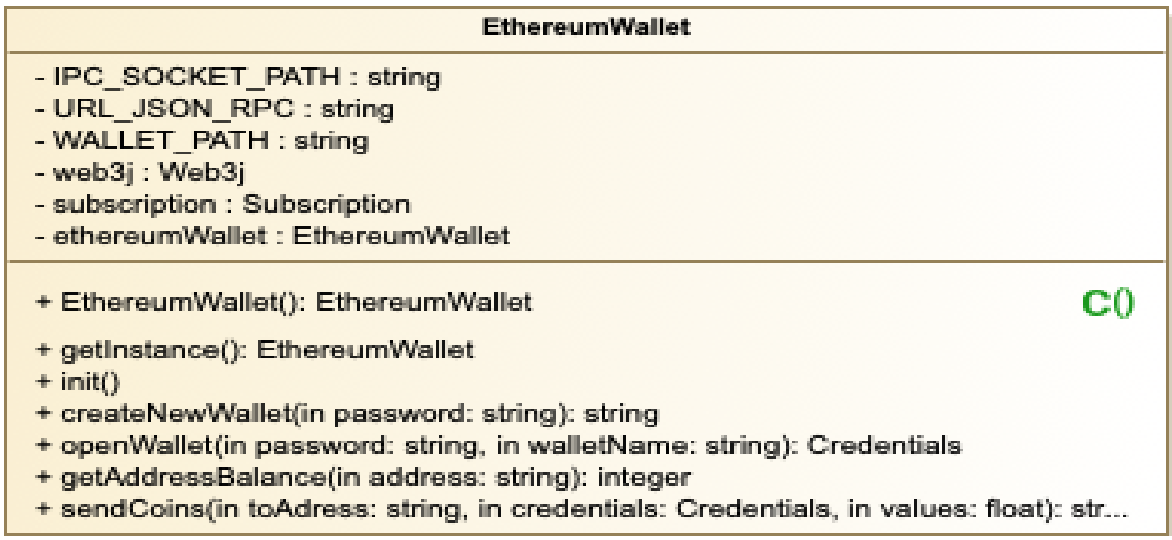


Figura 3.2: EthereumWallet

A continuación, se explica el funcionamiento de las funciones principales con partes del código de las mismas:

- *void init()*: Inicia una conexión con el nodo inicializado anteriormente, existen dos formas de hacerlo a través de IPC o por HTTP.

```
web3j = Web3j.build(new UnixIpcService(IPC_SOCKET_PATH));
web3j = Web3j.build(new HttpService(URL_JSON_RPC));
```

Las variables IPC_SOCKET_PATH y URL_JSON_RPC se inicializan a $\sim$/.*gophersland_ethereum_r1*/*geth.ipc* y http://127.0.0.1:8546 respectivamente.

- *String createNewWallet(String password)*: Permite crear una nueva cuenta en la red de pruebas de Ethereum, cuenta con un string de parámetro de entrada que será la contraseña para acceder a la cuenta y devuelve otro string, la dirección de la cuenta.

```
public String createNewWallet(String password) throws Exception {
  try {
      File file = new File(WALLET_PATH);
      String name = null;
      if (file.exists()) {
        name = WalletUtils.generateLightNewWalletFile(password, file);
        return name;
      } else {
        throw new Exception("Invalid WALLET_PATH " + WALLET_PATH);
      }

  } catch (Exception ex) {
      throw ex;
  }
}
```

- *Credentials openWallet(String password, String walletName)*: Obtiene las credenciales de una cuenta Ethereum, estas credenciales son necesarias para poder realizar transacciones desde dicha cuenta. Recibe como parámetro de entrada el nombre y la contraseña de la cuenta y devuelve las credenciales.

```
public Credentials openWallet(String password, String walletName)
throws Exception {
  Credentials credentials =
  WalletUtils.loadCredentials(password,walletName);
  return credentials;
}
```

- *BigInteger getAddressBalance(String address)*: Accede al balance de la cuenta cuya dirección se introduce como parámetro de entrada. Devuelve un entero que representa los WEI que contiene la cuenta, un Wei equivale a $10^{-18}$ Ether.

```
 public BigInteger getAddressBalance(String address)
```

```
 throws Exception {
  try {
      EthGetBalance ethGetBalance = web3j
        .ethGetBalance(address, DefaultBlockParameterName.LATEST)
        .sendAsync()
        .get();
      BigInteger wei = ethGetBalance.getBalance();
      return wei;
  } catch (Exception ex) {
      throw ex;
  }
}
```

- *String sendCoins(Credentials credentials, String toAddress, BigDecimal value)*: Envía un numero determinado de Wei a entre dos cuentas y devuelve un hash como comprobante de la transacción. Se necesita introducir como parámetros de entrada las credenciales de la cuenta que va hacer la transacción, la dirección de la cuenta receptora y el número de Wei que se quieren enviar.

```
public String sendCoins(Credentials credentials, String toAddress,
  BigDecimal value) throws Exception {
  try {
  TransactionReceipt transactionReceipt = Transfer.sendFunds(web3j,
  credentials, toAddress, value, Convert.Unit.ETHER);
  String transactionHash = transactionReceipt.getTransactionHash();
  return transactionHash;
  } catch (Exception ex) {
      throw ex;
  }
}
```

# Capítulo 4

# Información Semántica

Se va a representar un Smart Contract a través de una ontología, una ontología es una especificación formal y explícita de tipos, propiedades y relaciones entre tipos. Para crear las ontologías necesarias se ha utilizado la aplicación Protégé que se trata de un editor de ontologías. Y el lenguaje utilizado para definir la ontología será OWL2. Mediante esta ontología se pretende conseguir representar la información relevante para las dos partes que intervienen en un Smart Contract; a modo ilustrativo, se ha considerado una aplicación concreta como un contrato de compra-venta de entradas para un evento, pero sería fácilmente extrapolable a cualquier proceso de acuerdo entre dos partes a través de una o varias variables de rango numérico.

La ontología consta de dos entidades diferenciadas, un cliente y un vendedor. El vendedor, pone las condiciones de la venta de su producto, es decir, el precio y la cantidad que esta dispuesto a vender, así como una descripción del mismo e información referente a su producto.

Para ello, la clase vendedor cuenta con las siguientes propiedades:

- Direccion, hace referencia la dirección de la cuenta Ethereum del vendedor donde se enviarán las criptomonedas correspondientes una vez se haya hecho efectiva la cuenta.
- Tickets, número de entradas disponibles para el evento o el stock de un producto.
- Precio, cantidad de criptomonedas que vale el producto.
- Info, una breve descripción del evento o una url que contenga más información.
- Cod_verifcacion, una vez realizada la transferencia se obtiene un código hash de la misma que será almacenado para en el caso de haber una devolución poder comprobar que el usuario que la solicita efectivamente posee una producto verdadero.

El cliente por su parte debe ser capaz de aportar datos acerca de su balance para saber si puede o no adquirir el producto del vendedor. Para ello, la clase cliente cuenta con las siguientes propiedades:

- Direccion, que hace referencia a al dirección de la cuenta Ethereum desde la cual se transferirán las criptomonedadas para adquirir el producto.
- Balance, es la cantidad de criptomonedas que posee el cliente para poder comprar el producto.
- Tickets, el número de entradas que ya ha adquirido el cliente.
- Cod_verificacion, el código hash resultante de hacer la transferencia que será necesario en caso de que el usuario quiera hacer una devolución.

En la Figura 4.1 se observa una captura de pantalla de la aplicación Protégé con las propiedades de la ontolgía:

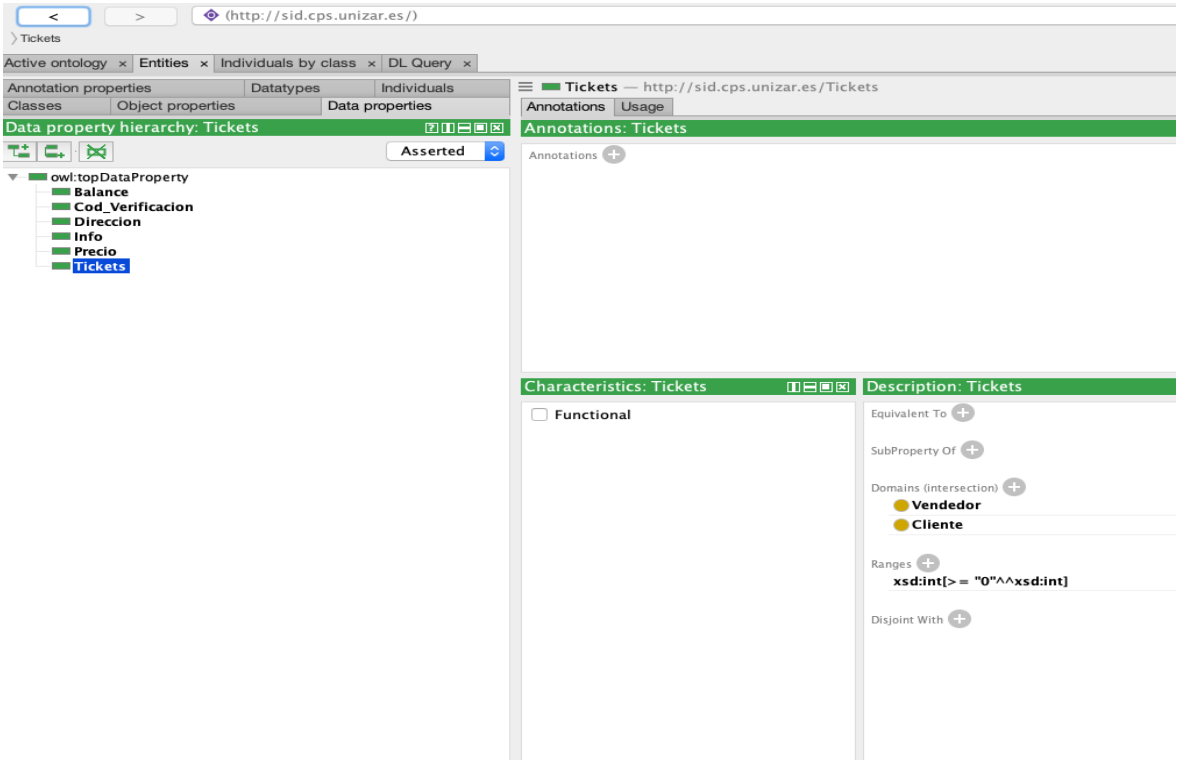


Figura 4.1: Propiedades de la ontología

Para poder trabajar en Java con la ontología se ha utilizado OWLAPI, que se trata de una API de Java que permite la creación, manipulación y serialización de ontologías en lenguaje OWL2. También se ha utilizado HermiT que es un razonador de ontologías en lenguaje OWL. Dado un fichero OWL, HermiT puede determinar si la ontología es consistente y recuperar valores de una propiedad de la ontología. [8]

En la Figura 4.2 se observa a estructura de la clase correspondiente que interactúa con la ontología:

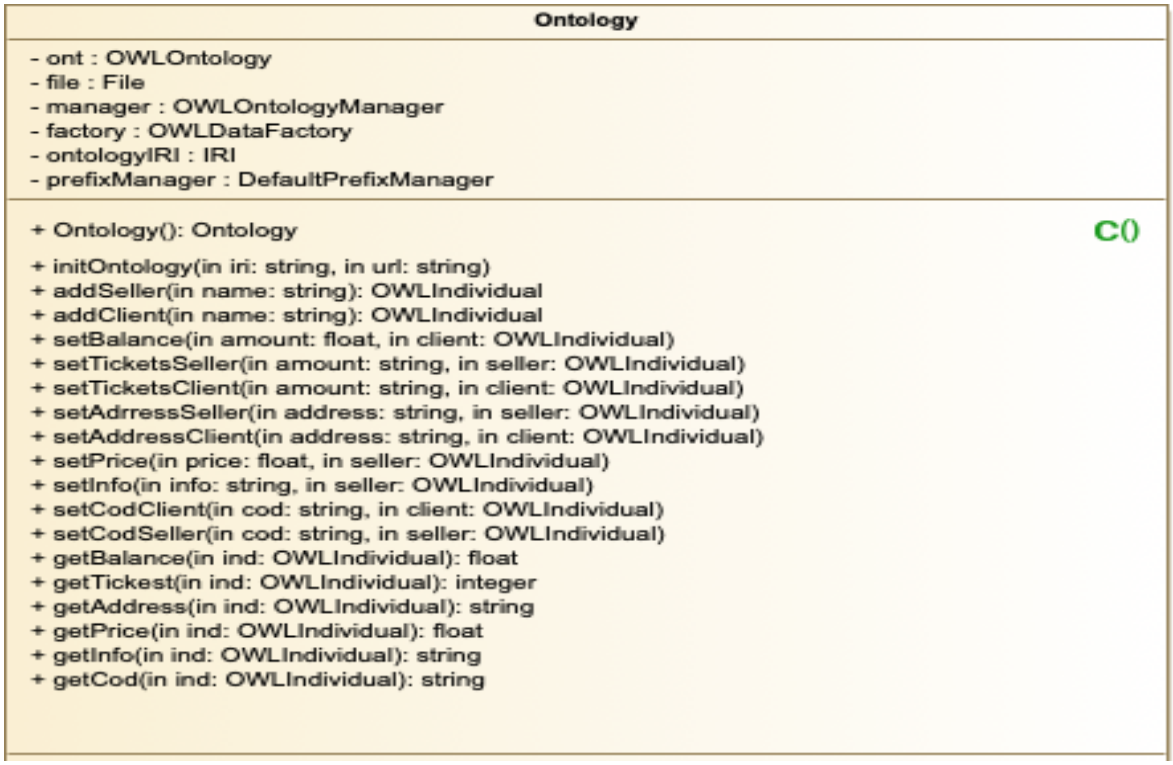


Figura 4.2: Ontology

Las principales funciones que componen la clase son las siguientes:

- *void initOntology(String iri, String url)*: carga la ontología a partir de un fichero OWL que se le pasa en el parámetro *url* y de una IRI que se le pasa en el parámetro *iri*.

```
public void initOntology(String iri, String url)
throws OWLOntologyCreationException {
    prefixManager = new DefaultPrefixManager();
    ontologyIRI = IRI.create(iri);
    file = new File(url);
    manager = OWLManager.createOWLOntologyManager();
    ont = manager.loadOntologyFromOntologyDocument(file);
    factory = manager.getOWLDataFactory();


  }
```

- *void setTicketsSellet(int amount, OWLIndividual seller)*: introduce el número de tickets que quiere poner el vendedor a la venta y actualiza la ontología. Existe una función *set* para cada una de las propiedades de la ontología.

```
public void setTicketsVendedor(int amount, OWLIndividual seller)
 throws OWLOntologyStorageException {

  OWLDataProperty tickets =
  factory.getOWLDataProperty("Tickets", prefixManager);

  OWLDataPropertyAssertionAxiom addProp =
  factory.getOWLDataPropertyAssertionAxiom(tickets,seller,amount);

  AddAxiom addAxioma3 = new AddAxiom(ont, addProp);

  manager.applyChange(addAxioma3);
  manager.saveOntology(ont,IRI.create(file.toURI()));
  }
```

- *int getTickets(OWLIndividual ind)*: devuelve el número de tickets que tiene un individuo perteneciente a la ontología para ello se utiliza el razonador para deducir el valor de la propiedad correspondiente, aunque no se encuentre explícitamente en la ontología. Existe una función *get* para cada una de las propiedades de la ontología.

```
  public int getTickets(OWLIndividual ind) {
    OWLDataProperty tickets =
    factory.getOWLDataProperty("Tickes", prefixManager);

    Set<OWLLiteral> values = ind.getDataPropertyValues(tickets, ont);

    String resul = values.toString();
    int i=resul.indexOf('\"');
    int j=resul.lastIndexOf('\"');
    resul.substring(i, j);
    int result = Integer.parseInt(resul);
    return result;

  }
```

El siguiente fragmento de código representa la compra de entradas de un evento, para ello es necesario comprobar el balance del comprador y las entradas disponibles por parte del vendedor.

```
int total=amount*ont.getPrecio(seller);
int available=ont.getTickets(seller);
if (ont.getBalance(client)>total && available>=amount) {
  org.web3j.crypto.Credentials cred=
  ethereumWallet.openWallet(password, ont.getAddress(client));
  BigDecimal ether=BigDecimal.valueOf(total);
  String hash=ethereumWallet.sendCoins(cred,ont.getAddress(seller),ether);
  ont.setTicketsClient(amount, client);
```

```
  ont.setTicketsSeller(available-amount, seller);
  ont.setCodClient(hash, client);
  ont.setCodSeller(hash, seller);
}else {
  System.out.println("No tienes saldo suficiente.");
}
```

En el caso de que hubiera mas restricciones para comprar el producto habría que combinar todas las condiciones de manera conjuntiva antes de proceder a la transacción. Previamente el razonador comprueba que la ontología es consistente, por lo que se verifica que las restricciones de las dos partes son compatibles.

# Capítulo 5

# Lógica difusa

La lógica difusa es una lógica que utiliza expresiones que no son ni totalmente ciertas ni completamente falsas, es decir, es la lógica aplicada a conceptos que pueden tomar un valor cualquiera de veracidad dentro de un conjunto de valores no binario que oscilan entre dos extremos, la verdad absoluta y la falsedad total. Conviene recalcar que lo que es difuso, borroso, impreciso o vago no es la lógica en sí, sino el objeto que estudia: expresa la falta de definición del concepto al que se aplica. La lógica difusa permite tratar información imprecisa, como estatura media o temperatura baja, en términos de conjuntos borrosos, es decir, que se puede pertenecer parcialmente. [6]

Según la teoría de la lógica clásica el conjunto "hombres altos" es un conjunto al que pertenecerían los hombres con una estatura mayor a un cierto valor, que se puede establecer en 1.80 metros, por ejemplo, y todos los hombres con una altura inferior a este valor quedarían fuera del conjunto. De este modo un hombre que mide 1.81 metros de estatura pertenecería al conjunto hombre altos, y en cambio un hombre que mida 1.79 metros de altura ya no pertenecería a ese conjunto. Sin embargo, no parece muy lógico decir que un hombre es alto y otro no lo es cuando su altura difiere en dos centímetros. El enfoque de la lógica difusa considera que el conjunto "hombres altos" es un conjunto que no tiene una frontera clara para pertenecer o no pertenecer a él: mediante una función que define la transición de "alto" a "no alto" se asigna a cada valor de altura un grado de pertenencia al conjunto, entre 0 y 1. Así por ejemplo, un hombre que mida 1.79 podría pertenecer al conjunto difuso "hombres altos" con un grado 0.8 de pertenencia, otro que mida 1.81 con un grado 0.85, y otro que mida 1.50 m con un grado 0.1. Visto desde esta perspectiva se puede considerar que la lógica clásica es un caso límite de la lógica difusa en el que se asigna un grado de pertenencia 1 a los hombres con una altura mayor o igual a 1.80 y un grado de pertenencia 0 a los que tienen una altura menor. [6]

En la Figura 5.1 se aprecian dos gráficas que representan como la lógica clásica y la lógica difusa representan el grado de pertenencia al grupo de "hombres altos" en

función a la estatura tal y como se explica en el párrafo anterior:

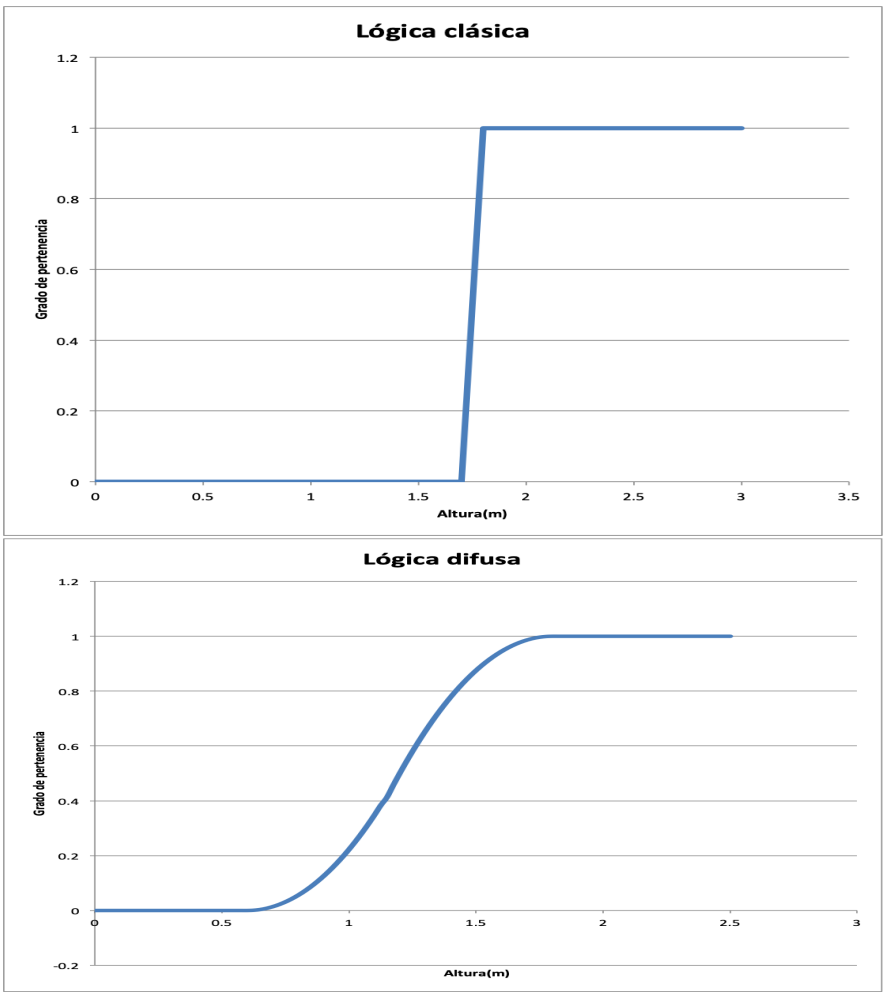


Figura 5.1: Gráfica lógicas

En este ámbito de los Smart Contracts es aplicable a negociaciones a la hora de acordar un precio para un producto. Por ejemplo, en el caso de compra-venta de un coche, el vendedor tiene un precio mínimo fijado del cual no puede bajar y, a partir de ahí, cuanto mayor es el precio, mayor es su satisfacción con la venta. En el caso del cliente que cuenta con un presupuesto máximo para gastarse en el coche, cuanto menor sea el precio del coche mayor será su grado de satisfacción. Con el sistema utilizado se calcula el máximo grado de satisfacción conjunta, este se obtine del punto de corte de las funciones que representan el grado de satisfacción por parte del vendedor y comprador en función del precio del procudto.

En la figura 5.2 se observa el caso en el que un vendedor tiene como máximo 1000 y mínimo 600 y un comprador cuyo máximo es 1000 y su mínimo 500. El grado de satisfacción del vendedor decrece cuanto menor es el precio de la venta sin embargo el del comprador aumenta cuento menor es el precio. El punto de corte se haya para un grado de satisfacción de 0,44, siendo este el máximo grado de satisfacción conjunta y

un precio de 777,77.

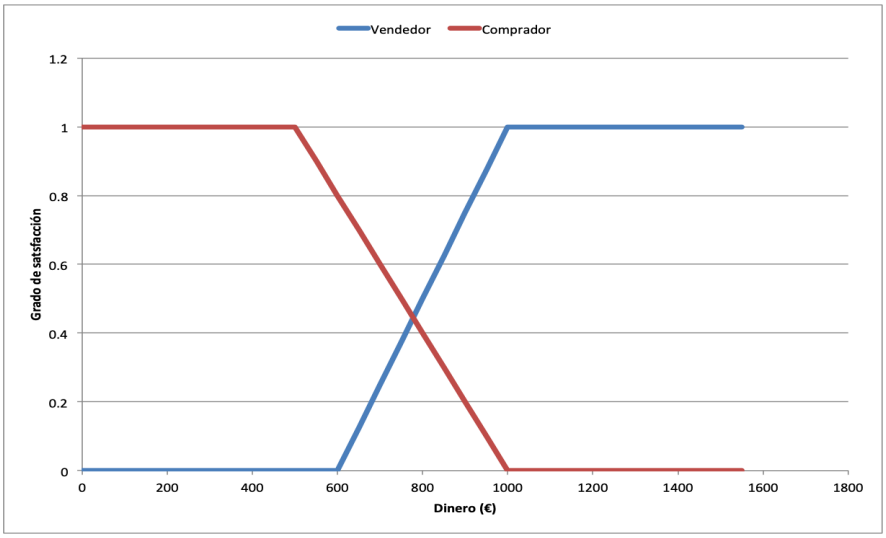


Figura 5.2: Gráfica comprador-vendedor

## 5.1. FuzzyDL

Para poder llevar a cabo esta parte del trabajo se ha utilizado el razonador para ontologías difusas FuzzyDL. Se ha utilizado Fuzzy OWL2 como lenguaje para programar ontologías difusas.

Para extender una ontología clásica a una difusa con Protégé es necesario crear un DataType y en la parte de anotaciones introducir las funciones correspondientes, para el caso de compra-venta se crean dos DataType, SellerPrice para el vendedor y BuyerPrice para el cliente. En la Figura 5.3 y en la Figura 5.4 se pueden observar las anotaciones correspondientes a BuyerPrice y SellerPrice respectivamente:

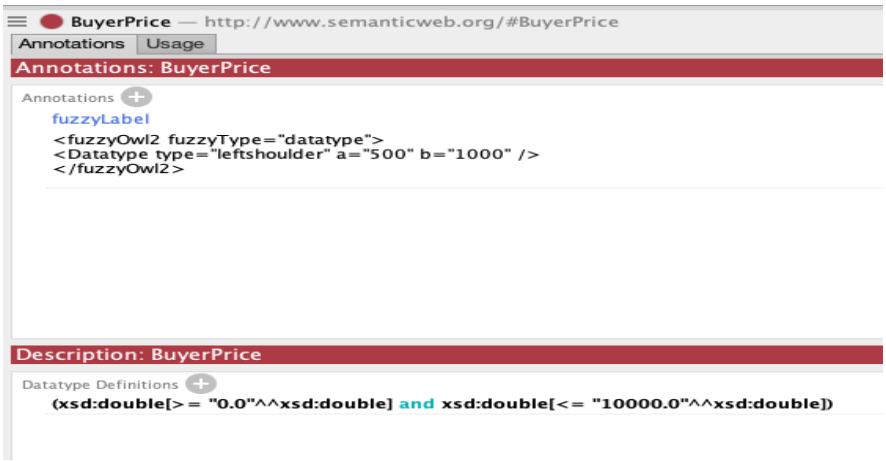


Figura 5.3: Anotación para BuyerPrice


Figura 5.4: Anotación para SellerPrice

Para poder interactuar con esta ontología se usó la API de Java para FuzzyDL, consiguiendo así obtener le grado de satisfacción de la operación de compra-venta. Sin embargo, se encontró una limitación al no poder obtener los valores para los que se obtenía el grado de satisfacción. Por lo tanto, se tuvo que modificar las funciones correspondientes de FuzzyDL para permitir está opción.

La forma de trabajar con la ontología es diferente que con la OWL API y el razonador Hermit visto en el capítulo anterior, tanto el cargar la ontología como la consulta que se necesita resolver. A continuación, se explica los pasos que se han seguido para lograr los acuerdos parciales:

1. Se configura el razonador y se crea el *KnowledgeBase* con la ontología que se encuentra en el *inputFile*. Se añade la propiedad responsable de llegar al grado de satisfacción óptimo por parte de las dos partes a las variables que se pueden mostrar, en este caso la propiedad es *hasPrice*

```
// Se cargar las opciones del razonador, usando el archivo "CONFIG"
ConfigReader.loadParameters("CONFIG", new String[0]);

KnowledgeBase kb = Parser.getKB(inputFile);
kb.milp.showVars.addConcreteFillerToShow("hasPrice");
```

2. Se realiza la consulta para encontrar el máximo grado de satisfacción entre el comprador y el vendedor.

```
// After having created KB and queries, start logical inference
kb.solveKB();

Concept c1 = kb.getConcept("Buyer");
Concept c2 = kb.getConcept("Seller");
```

```
Concept c = Concept.gAnd(c1, c2);

MaxSatisfiableQuery q = new MaxSatisfiableQuery(c);
Solution result = q.solve(kb);
```

3. Si el resultado es consistente se muestra por pantalla el grado de satisfacción alcanzado y el valor de la propiedad con el que se alcanza.

```
if (result.isConsistentKB())
{
System.out.println("Satisfiability degree: "+result.getSolution());
Hashtable<String, Double> vars = result.getShowedVariables();
for (String varName : vars.keySet())
  if (!varName.startsWith("("))
    System.out.println("Value of "+varName+" : "+vars.get(varName));
}
else
  System.out.println("KB is inconsistent");
```

Para el BuyerPrice y el SellerPrice correspondiente a las anotaciones de las figuras 5.3 y 5.4, el resultado obtenido es un grado de satisfacción de 0,44 para un precio de 777,77, tal como se aprecia en la Figura 5.5:

```
Satisfiability degree: 0.4444
Value of hasPrice(i1): 777.7778
```

Figura 5.5: Resultado

# Capítulo 6

# IPFS

Una vez realizado el Smart Contract en OWL2 se plantea el problema de dónde almacenarlo para que no pueda ser modificado. Debido a que se puede tratar de ficheros relativamente grandes para el tamaño de un bloque no es posible almacenarlos de manera eficaz en Ethereum por eso se obto por almacenarlos en IPFS.

En lugar de descargar archivos de servidores individuales como en HTTP, en IPFS, Sistema de archivos interplanetario, el usuario solicita a sus pares en la red que le proporcionen una ruta de acceso a un archivo. Esta condición permite la distribución de datos a gran escala con alta eficiencia, versiones históricas, contenido asegurado y verificado a través de hashes criptográficos. IPFS conecta todos los equipos informáticos con el mismo sistema de archivos creando una red distribuida de nodos, P2P. En términos sencillos, actúa de manera similar al protocolo BitTorrent, salvo que, en lugar de compartir e intercambiar archivos, IPFS intercambia objetos git. Esto quiere decir que todo el sistema se basa en un simple almacén de datos valor–clave. [9]

Es posible insertar cualquier tipo de contenido en el sistema. El usuario obtiene una clave que puede ser utilizada para recuperar el contenido en cualquier momento, dicha clave es totalmente independiente del origen de la información. Por lo tanto, IPFS realiza un direccionamiento del contenido en lugar del direccionamiento de la ubicación realizado por HTTP.

Cada archivo y todos los bloques dentro de este reciben un hash criptográfico que actúan como un identificador único. Los archivos duplicados se eliminan de la red y se realiza un seguimiento del historial de versiones de cada archivo. Esto se traduce en un contenido siempre disponible.

El almacenamiento de archivos grandes en Blockhain es complicado debido a la limitación de espacio de cada bloque de la cadena, sin embargo si que es factible su combinación con IPFS. En vez de almacenar en cada bloque transacciones verificadas es podría almacenar el hash referente a un archivo almacenado en IPFS.

Para poder trabajar en Java con IPFS se ha tenido que añadir unas dependencias

al proyecto de Eclipse:

```
<repositories>
   <repository>
       <id>jitpack.io</id>
       <url>https://jitpack.io</url>
   </repository>
 </repositories>

 <dependencies>
   <dependency>
     <groupId>com.github.ipfs</groupId>
     <artifactId>java-ipfs-api</artifactId>
     <version>v1.2.0</version>
   </dependency>
 </dependencies>
```

Es necesario instalar primero IPFS en la máquina que se va utilizar, para ello se han ejecutado los siguientes comandos:

1. Después de descargar el repositorio de https://ipfs.io/, lo primero es descomprimir el archivo e instalarlo.

```
$ tar xvfz go-ipfs.tar.gz
$ cd go-ipfs
$ ./install.sh
```

2. Una vez instalado se puede inicializar.

```
$ ipfs init
```

   El primer comando inicializa IPFS y proporciona un id de par en este caso: QmS4ustL54uo8FzR9455qaxZwuMiUhyvMcX9Ba8nUH4uVv.

3. Después de completar los pasos anteriores sería posible subir archivos a la red IPFS con el siguiente comando.

```
$ ipfs init add "ruta del archivo que se quiere subir"
```

La clase implementada en Java permite conectarse al nodo que se ha inicializado anteriormente, así como poder subir archivos y descargar archivos que ya estén almacenados en la red. En la Figura 6.3 se observa la estructura de la clase implementada.

A continuación, se explica el funcionamiento de las funciones principales:

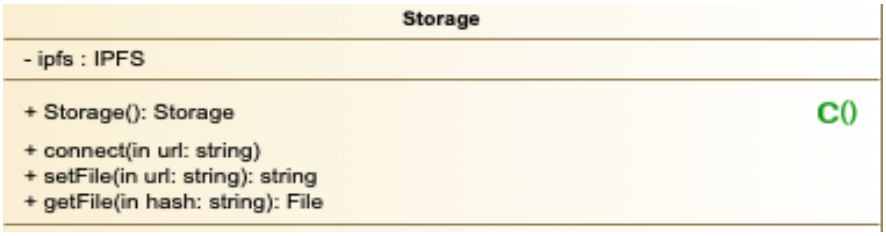


Figura 6.1: Storage

- *void connect(String url)*: Permite conectarse al nodo previamente inicializado pasándole un string con la url de dicho nodo.
- *String setFile(String url)*: Almacena el archivo de la ruta que se incluye como parámetro de entrada en IPFS y devuelve el hash relativo a ese archivo.
- *File getFile(String hash)*: Recupera archivos previamente almacenados en IPFS introduciendo como parámetro de entrada el hash referente al archivo.

# Capítulo 7

# Conclusiones

Tras el trabajo realizado se puede concluir que es posible la creación de Smart Contracts utilizando información semántica para ello es necesario utilizar distintas tecnologías, Ethereum, OWL2, IPFS, FuzzyDL o HermiT. Así mismo se ha comprobado que es posible alcanzar acuerdos parciales utilizando la lógica difusa, algo que no permiten los Smart Contracts clásicos programados en Solidity.

Para obtener estos resultados se han realizado las siguientes tareas:

1. Crear una Blockchain clásica para probar y entender su funcionamiento.

2. Inicializar un nodo de la red de prueba de Ethereum, Rinkeby, e implementar una clase en Java para poder interactuar con el nodo y llevar a cabo transacciones entre las diferentes cuentas.

3. Crear una ontología capaz de representar las dos partes que intervienen en un contrató y desarrollar una clase en Java que permitiera modificaciones sobre la ontología a través de OWLAPI.

4. Ampliar la ontología clásica del anterior apartado para que utilizará lógica difusa y conseguir acuerdos parciales a través de Fuzzy OWL2 y crear una clase en Java que se comunicará con la ontología para lo cual ha sido necesaria la API de FuzzyDL.

5. Inicializar un nodo en la red IPFS y crear una clase en Java que permitiera conectarse al nodo y almacenar archivos y recuperarlos. Esto es necesario para el almacenamiento de los Smart Contracts.

Con este trabajo se ha podido profundizar en la tecnología Blockchain observando sus aplicaciones reales y comprendiendo su funcionamiento. También se ha descubierto una nueva aplicación a la información semántica ampliando así los conocimientos obtenidos sobre este tema en la asignatura de Recuperación de la Información. Otras

asignaturas que también han sido útiles para este trabajo han sido Ingeniería del Software y Seguridad Informática.

Una posible continuación de este trabajo podría ser la implementación de un market place que funcionara con Smart Contracts en el que se pudiera adquirir diferentes entradas para eventos, conciertos de música, eventos deportivos, entradas de cine, etc. Aunque no solo sería aplicable al ámbito de compra venta, como ya se ha comentado anteriormente la ontología que representa el Smart Contract se puede extrapolar a cualquier tipo de acuerdo. Dos posibles aplicaciones muy interesantes podrían ser la realización de un contrato de trabajo o la asignación de créditos universitarios conforme se van superando las asignaturas correspondientes.

# Bibliografía


[1] Boris Motik Giorgos Stoilos Zhe Wang Birte Glimm, Ian Horrocks. Hermit: An owl 2 reasoner. *Journal of Automated Reasoning*, 53(3):245–269, 2018.

[2] CCN. 100 %: Dubai will put entire land registry on a blockchain. `https://www.ccn.com/100-dubai-put-entire-land-registry-blockchain/`, último acceso el 24/06/19.

[3] CCN. Japan could place its entire property registry on a blockchain. `https://www.ccn.com/japan-place-entire-property-registry-blockchain/`, último acceso el 24/06/19.

[4] Heraldo de Aragón. La DGA impulsará un proyecto de contratación pública utilizando 'blockchain'. `https://www.heraldo.es/noticias/aragon/2018/03/07/la-dga-impulsara-proyecto-contratacion-publica-utilizando-blockchain-1228833-300.html`, último acceso el 24/06/19.

[5] DiarioBitcoin. Ethereum 101 – ¿qué es ethereum? `https://www.diariobitcoin.com/index.php/ethereum-101-que-es-ethereum/`, último acceso el 24/06/19.

[6] Logica difusa. `https://www.tdx.cat/bitstream/handle/10803/6887/04Rpp04de11.pdf`, último acceso el 24/06/19.

[7] Ethereum. `https://www.ethereum.org/`, último acceso el 24/06/19.

[8] HermiT. `http://www.hermit-reasoner.com/`.

[9] IPFS. `https://ipfs.io/`, último acceso el 24/06/19.

[10] I. Sylla N. Fairoza O. Choudhury, N. Rudolph and A. Das. Auto-generation of smart contracts from domain-specific ontologies and semantic rules. 2018.

[11] Héctor E. Ugarte R. A more pragmatic web 3.0: Linked blockchain data. 2018.

[12] Solidity. `https://solidity-es.readthedocs.io/es/latest/`, último acceso el 24/06/19.

[13] A. Third and J. Domingue. Linked data indexing of distributed ledgers. *LD-DL@WWW 1st International Workshop on Linked Data and Distributed Ledgers*, 2017.

[14] Web3j. Biblioteca. `https://github.com/web3j/sample-project-gradle`, último acceso el 24/06/19.

[15] Web3j. Documentación. `https://docs.web3j.io/`, último acceso el 24/06/19.

[16] Wikipedia. Bitcoin. `https://es.wikipedia.org/wiki/Bitcoin`, último acceso el 24/06/19.

[17] Wikipedia. Cadena de bloques. `https://es.wikipedia.org/wiki/Cadena_de_bloques`, último acceso el 24/06/19.

[18] G Wood. Ethereum: A secure decentralised generalised transaction ledger. 2015.

[19] Xataka. Qué es blockchain: la explicación definitiva para la tecnología más de moda. `https://www.xataka.com/especiales/que-es-blockchain-la-explicacion-definitiva-para-la-tecnologia-mas-de-moda`, último acceso el 24/06/19.

# Anexo A

# Diagrama de Gantt