¿ Qué es Ethereum ?

Ethereum es un **Blockchain** o **Tecnología de Contabilidad Distribuida (Distributed Ledger Technology),** también conocida por sus siglas **DTL**. Es una **red peer to peer**, donde todos los **"peer"** almacenarán la misma **copia de la base de datos** en el **blockchain (una base de datos distribuida)** y ejecuta una **Máquina Virtual Ethereum (EVM)** para mantener y alterar su estado. **Proof of Work (PoW)** está integrado en la tecnología **blockchain** por lo que la creación de un **nuevo bloque** requiere que todos los **miembros de la red** validen la **prueba de trabajo (PoW)**. El consenso se logra a través de incentivación de los **mineros** para aceptar siempre la **cadena más larga** de bloques en el **blockchain** para poder ganar un **token criptográfico** de valor: **"ether"**.

Diferencias con Bitcoin

Ethereum es un **blockchain programable**. En lugar de dar a los usuarios un conjunto de operaciones predefinidas (transacciones **Bitcoin**), **Ethereum** permite a los usuarios crear sus propias operaciones de cualquier complejidad así lo desean. De este modo, sirve como una **plataforma** para diferentes tipos de **aplicaciones** de **blockchain descentralizadas** y no limitado a **criptomonedas**.

La plataforma Ethereum comprende.

1. un lenguaje de programación integrado **Turing completo**.
2. una **computadora blockchain**, que permite que cualquier miembro participante pueda escribir **contratos inteligentes** y aplicaciones descentralizadas simplemente escribiendo la lógica en unas pocas líneas de código.
3. un **protocolo de red peer to peer (P2P)**
4. su **seguridad** está **garantizada** gracias a una **fuerte criptografía.**

Consecuencias

Ethereum es una **computadora mundial,** que genera una **paralelización masiva** de toda la información a través de su **red**, no con el fin de ser más eficiente, en realidad es más lenta y cara que un ordenador tradicional. Por lo tanto, cada **nodo Ethereum** corre en el **EVM** con el fin de mantener el **consenso** a través de la **blockchain**. **Consenso descentralizado** que permite a **Ethereum** niveles extremos de **tolerancia a fallos**, elimina los **tiempos de inactividad**, y hace que los **datos almacenados** en el **blockchain sean inalterables y resistentes a la censura**.

Actividades donde puede utilizarse Ethereum

Bitcoin permite a las personas intercambiar **dinero** en **efectivo** sin la intervención de ningún intermediario como **instituciones financieras, bancos o gobiernos**. El impacto de **Ethereum** tendrá un **mayor alcance**. Hay tres tipos de aplicaciones que pueden implementarse en la capa superior de **Ethereum**.

1. **Aplicaciones financieras,** que proporcionan a los usuarios formas más poderosas de administrar y gestionar los contratos usando su dinero. Esto incluye sub-monedas, derivados financieros, contratos de cobertura, carteras de ahorro, testamentos, y en última instancia, incluso algunos tipos de contratos de trabajo. Intercambios **financieros** de cualquier complejidad podrían llevarse a cabo de forma automática y fiable utilizando el **código** que se ejecuta en **Ethereum**.
2. **Aplicaciones semi-financieras,** donde el dinero está involucrado pero hay un lado no monetario de peso; un ejemplo perfecto de esto son las recompensas por la solución de problemas computacionales.
3. **Aplicaciones no financieras,** Podrá actuar en cualquier **escenario** donde la **confianza, la seguridad y la permanencia** son **importantes,** como ser **activos-registros**, **sistemas de votación (voto online)**, **gobierno descentralizado**, **administración pública e internet**.

¿Cómo funciona Ethereum?

Ethereum posee muchas de las características y tecnologías que son familiares para los usuarios de **Bitcoin**, al mismo tiempo que introduce muchas modificaciones e innovaciones propias. Considerando que el **blockchain Bitcoin** es básicamente una **lista de transacciones**, la **unidad básica** del **Ethereum** son **cuentas**. El **Ethereum blockchain** hace un seguimiento del **estado** de cada **cuenta**, ya que todas las **transiciones de estado** en la **blockchain Ethereum** son **transferencias de valor** e **información** entre **cuentas**.

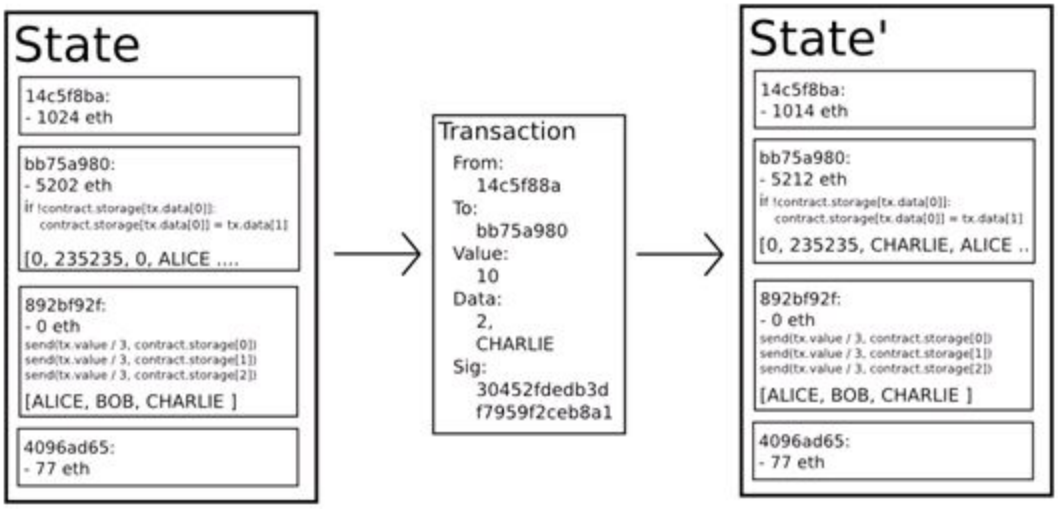
Hay dos tipos de cuentas:

1. **Externally Owned Account (EOA)**, las mismas son controladas por claves privadas. Los usuarios controlan **EOA**, **ya que pueden controlar las claves privadas que dan el control sobre una EOA**.
2. **Contract Accounts**, que son controladas por su **código de contrato** y sólo pueden ser **activadas** por un **EOA.** Las **cuentas de contrato**, por el contrario, se rigen por su **código interno**. Si están **controlados** por un usuario, es porque

están **programados** para ser controlado por un **EOA** con una cierta dirección, que es a su vez controlada por quien tiene las **llaves privadas** que controlan a su vez la **EOA**.

El término popular **contratos inteligentes** se refiere al **código** de una cuenta de licitación –**programas** que se ejecutan cuando una **transacción** se envía a esa cuenta. Los usuarios pueden crear **nuevos contratos** mediante la implementación de **código en blockchain**. Al igual que en **Bitcoin**, los usuarios deben **pagar pequeñas cuotas** de **transacción** a la red. Esto protege al **blockchain Ethereum** de tareas computacionales frívolas o maliciosas, como los ataques **DDoS** o **bucles infinitos**. El remitente de una transacción debe pagar por cada paso del **programa** para poder activarlo, incluyendo computación y almacenamiento en la memoria. Estas tasas se pagan en **ether de Ethereum.** Estas tasas de transacción son recogidas por los **nodos que validan la red**. Estos **mineros** son **nodos de la red Ethereum** que reciben pagos por **verificar** y ejecutar **transacciones**. Los **mineros** después agrupar las **transacciones,** que incluyen muchos cambios en el **estado** de las **cuentas,** en el **blockchain Ethereum** en lo que llamamos **bloques**, a continuación, ellos **compiten** entre sí para que su bloque sea el próximo en añadirse al **blockchain**. Los **mineros** son recompensados con **ether** por **cada bloque** creado con **éxito**. Esto proporciona el **incentivo económico** para las personas que dedican recursos en **hardware** y **electricidad** a la **red Ethereum**. Al igual que en la **red Bitcoin**, los **mineros** tienen la tarea de **resolver un problema matemático complejo** con el fin de **crear un bloque**. Esto se conoce como una **prueba de trabajo (PoW)** . Cualquier **problema computacional** que requiera **órdenes** de magnitud y **recursos** para resolver los **algoritmos** que se necesitan para **verificar la solución,** es un buen candidato para la **prueba de trabajo**. Con el fin de desalentar la **centralización** debido al uso de **hardware especializado (ASIC)**, como ha ocurrido en la **red Bitcoin**, **Ethereum** eligió un **problema computacional** que requiere **memoria** y **CPU**. El **hardware** ideal, es de hecho, un **ordenador común**.

Desde un punto de vista técnico, el **libro contable** de una **criptomoneda** como **Bitcoin**, puede ser visto como un **sistema de transición de estados**, donde hay un **estado inicial** (que consiste en el estado de propiedad de todos los Bitcoin existentes) y una **función de transición de estado** que toma un **estado inicial** y una **transacción** y emite un **nuevo estado**, que es el resultado.



APPLY(S, TX) -> S' o ERROR

APPLY({ Alice: $ 50, Bob: $ 50 },” enviar $ 20 de Alice a Bob”) = { Alice: $ 30, Bob: $ 70 }

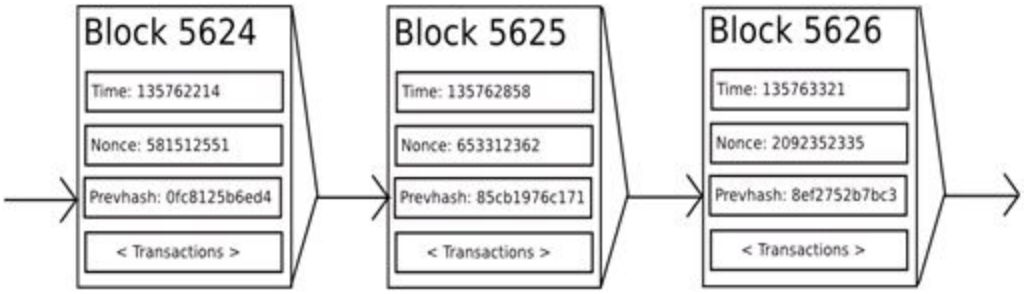
APPLY({ Alice: $ 50, Bob: $ 50 },” enviar $ 70 de Alice a Bob”) = ERROR

State. El **estado** en **Bitcoin** es la colección de todas las monedas **UTXOutput (UTXO)** que han sido acuñadas y todavía no gastadas, cada **UTXO** tiene una denominación y un propietario (definida por una **dirección de 20 bytes** que es esencialmente una **clave criptográfica pública**). Una **transacción** contiene una o más entradas, cada entrada contiene una referencia a una **UTXO** existente y una **firma criptográfica** producida por la **clave privada** asociada con la **dirección del propietario**, y una o más salidas, cada salida contiene una **nueva UTXO** que se añadirá al **estado**.

Función APPLY(S, T) -> S'. Es una **función de transición de estado** y puede definirse aproximadamente del modo siguiente: Por cada entrada en **TX:** Si la **UTXO** referenciada no está en **S**, devolver **ERROR**. Si la firma proporcionada no coincide con el propietario de la **UTXO**, devolver **ERROR**. Si la suma de todas las entradas **UTXO** es menor que la suma de todas las salidas **UTXO**, devolver **ERROR**. Devolver **S** con todas las **entradas UTXO eliminadas** y con todas las **salidas UTXO añadidas**.

La primera mitad de la primera etapa, impide al remitente de la transacción, gastar monedas que no existen, mientras que la segunda mitad de la primera etapa, impide al remitente de la transacción gastar las monedas de otras personas. El segundo paso sirve para hacer cumplir la conservación del valor.

Minería



Si tuviéramos acceso a un servicio centralizado de confianza, este sistema sería fácil de implementar; simplemente podría codificarse exactamente como se describe, usando el **disco duro** de un **servidor centralizado** para realizar un seguimiento de los **estados**. Sin embargo, con **Bitcoin**, lo que estamos tratando de construir, es un **sistema de moneda descentralizada**, por lo que tendremos que combinar:

1. **un sistema para controlar el estado de las transacciones**
2. **un sistema de consenso, con el fin de garantizar que todo el mundo está de acuerdo en el orden de las operaciones. El proceso de consenso descentralizado de Bitcoin, requiere de nodos de red para intentar producir, continuamente, paquetes de transacciones denominados "bloques".**

La red está diseñada para crear más o menos **un bloque cada diez minutos**, de modo que cada bloque contiene un **sello de tiempo**, y **una referencia (hash)** al bloque anterior, así como una lista de t**odas las transacciones** que se han producido desde el **último bloque**. Con el tiempo, se va creando una mayor **cadena de bloques o "blockchain" persistente**, que se actualiza constantemente para representar el último estado del libro contable de **Bitcoin**.

El algoritmo para comprobar si un bloque es válido.

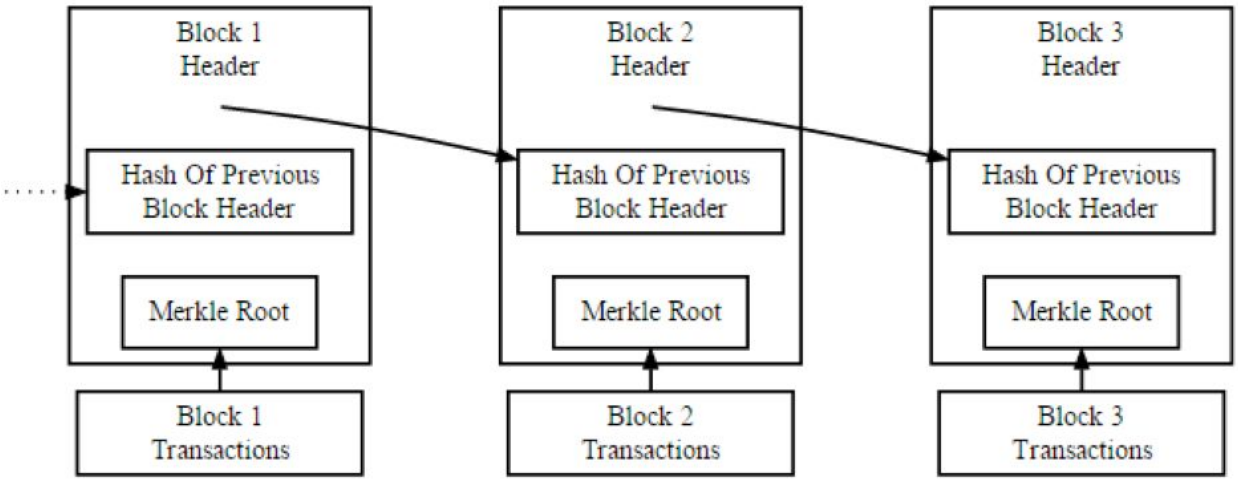
Está expresado en el siguiente paradigma:

1. Comprueba si el bloque anterior referenciado por el bloque actual existe y es válido.
2. Comprueba que el **sello de tiempo** del bloque actual es mayor que el del bloque previo pero menor de 2 horas que el futuro bloque.
3. Comprueba que la **prueba de trabajo** del **bloque es válida**.
4. **S[0]** es el estado final del bloque anterior. Supongamos que **TX** es la **lista de transacciones del bloque** con **n** Para todo **i** en **0... n-1**, podemos establecer

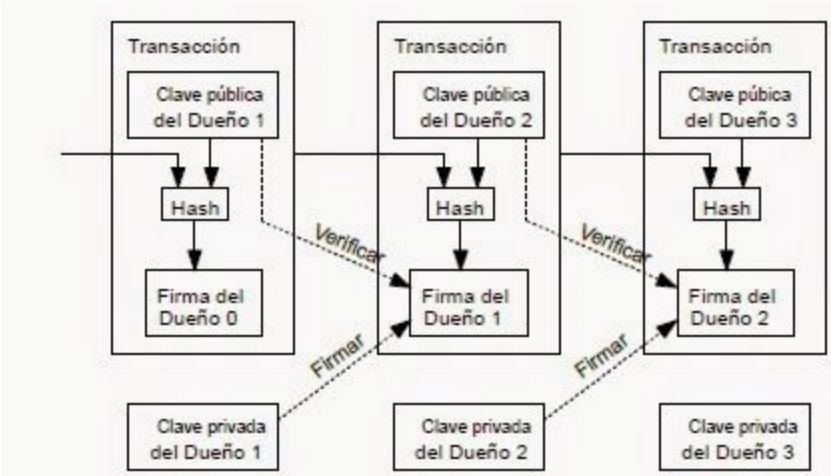
S[i+1] = APPLY(S[i], TX[i]). Si alguna transacción **devuelve un error**, sale y devuelve **falso (false)**.

5. Retorna **verdadero (true)**, y registra **S[n]** cómo el **estado final** de este **bloque**.

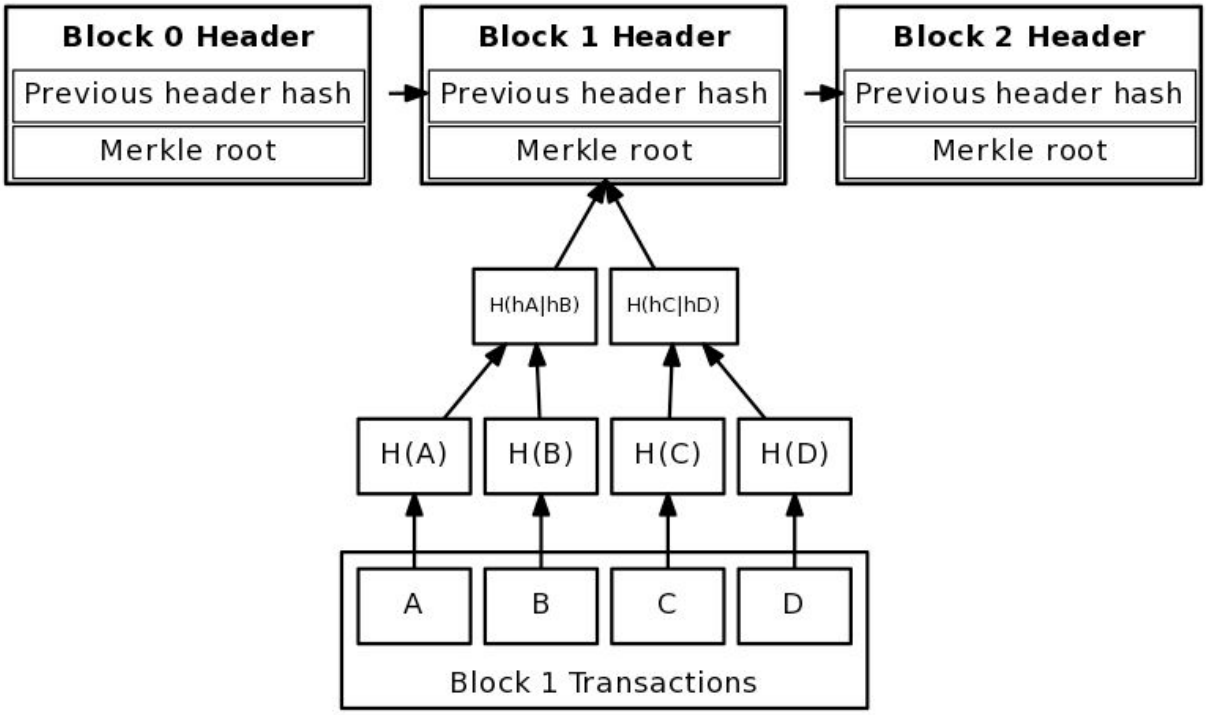
En esencia, cada transacción dentro del bloque, debe proporcionar una transición de estado válida desde lo que fue su estado original antes de que la transición fuera ejecutada, hasta el nuevo estado. Observamos, que el estado no está codificado en el bloque de ninguna manera; es solamente una abstracción recordada por el nodo de validación y sólo puede (de forma segura) ser calculada, para cualquier bloque, partiendo del estado génesis y aplicando secuencialmente cada transacción a cada bloque. Además, el orden en que el minero incluye las transacciones dentro del bloque importa, es decir, si hay **dos transacciones A** y **B** en un **bloque**, tal que **B** gasta un **UTXO** creada por **A**, entonces el **bloque será válido** si **A** está antes que **B**, pero no de otra manera.



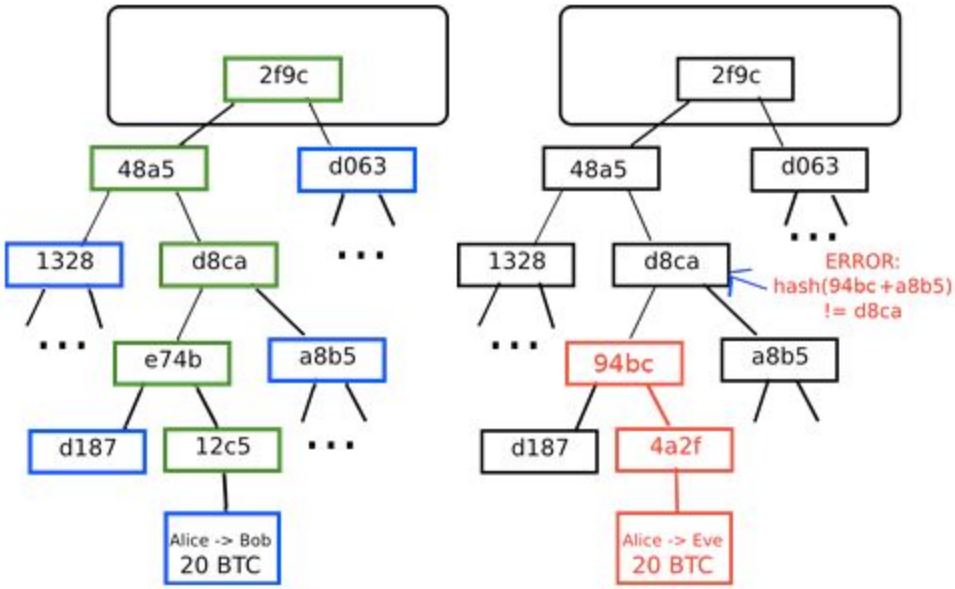
Simplified Bitcoin Block Chain



Árboles de Merkle



Merkle tree connecting block transactions to block header merkle root



En la parte izquierda: es suficiente con presentar un **pequeño número de nodos** en un **árbol de Merkle** para dar una prueba de la **validez de la rama.**

En la parte Derecha: cualquier intento de **cambiar** una parte de un **árbol de Merkle** finalmente dará una **inconsistencia** en algún lugar de la **cadena**.

Una característica importante de la **escalabilidad en Bitcoin** es que los bloques se almacenan en una **estructura de datos de múltiples niveles**. El **hash** de un bloque es solamente el **hash** de la **cabecera del bloque**, un fragmento de datos de unos **200 bytes** que contiene la **fecha** y **hora** (**time stamp**), el **nonce** (**en criptografía se llama nonce a un número arbitrario que solamente se utiliza una única vez al realizar algún tipo de operación**), el **hash** del **bloque anterior** y el **hash raíz de una estructura de datos llamada árbol de Merkle** que almacena todas las **transacciones del bloque**. Un **árbol de Merkle** es un tipo de **árbol binario**, compuesto por un conjunto de **nodos** con un gran número de **nodos hoja** en la parte **inferior del árbol** que contiene los **datos subyacentes**, un conjunto de **nodos intermedios** donde cada **nodo es el hash de sus dos hijos**, y finalmente un **único nodo raíz**, también formado por el **hash de sus hijos**, que representa la parte **superior (top) del árbol**. El objetivo de los **árboles de Merkle** es permitir que los **datos de un bloque** puedan ser entregados por partes: un **nodo puede descargar solamente la cabecera de un bloque** desde una fuente y una pequeña parte del **árbol relevante para él**, desde otra fuente, y todavía asegurar que los datos son correctos. La razón por lo que esto funciona es porque los **hashes** se **propagan** hacia arriba: si un usuario malintencionado intenta hacer un cambio en una **transacción falsa** en la parte inferior del **árbol de Merkle**, este cambio provocará un cambio en el **nodo superior** y seguidamente otro cambio en el nodo superior a este, hasta que finalmente, se produzca un **cambio en la raíz del árbol** y por tanto en el **hash del bloque,** haciendo que el protocolo tenga que registrarlo como un bloque completamente diferente (y casi con toda seguridad con una prueba de trabajo inválida). El **protocolo de árbol de Merkle** es, sin duda esencial para la sostenibilidad a largo plazo. Un **nodo completo** en la **red Bitcoin**, que almacena y procesa la totalidad de cada bloque, requiere unos **15 GB** de espacio en el disco (Abril de 2014), esta cantidad que está creciendo a un ritmo de **un gigabyte** al **mes**. Actualmente, esto es viable para algunos ordenadores de escritorio pero no para móviles, en el futuro solo empresas y aficionados serán capaces de poder participar en el blockchain. El **protocolo** conocido como **verificación del pago simplificado (simplified payment verification o SPV)** permite que otra clase de nodos puedan existir, los llamados **nodos ligeros o light nodes**, que descargan los encabezados de los bloques, realizan la verificación de la **prueba de trabajo** en éstos, y luego descargan solamente las **ramas** asociadas a las transacciones que son relevantes para ellos. Esto permite que los nodos ligeros determinar con una fuerte garantía de seguridad cual es el estado de

cualquier transacción **Bitcoin**, y sus balances actuales, descargando solamente una porción muy pequeña de toda la cadena de bloques.

Usos de la plataforma Ethereum

Sistema de Testigos (Tokens). Un sistema de **testigos (tokens)** construido sobre la **cadena de bloques** tiene un rango de aplicaciones muy amplio, que van desde las **sub-monedas** que pueden representar activos como el **dólar** o el **oro**, hasta **acciones de empresas**, **testigos individuales** que representan **propiedad inteligente**, **cupones seguros infalsificables** e incluso sistemas de **testigos simbólicos**, sin vínculos con un valor convencional y que pueden usarse como sistemas de incentivación. Los **sistemas de testigos** son sorprendentemente fáciles de implementar en **Ethereum**.

Derivados Financieros y Monedas de Valor Estable. Los derivados financieros son la aplicación más común de un **contrato inteligente** y uno de los más simples de implementar en código. El principal desafío que entrañan su implementación es que la mayoría de ellos requieren una **referencia a un precio externo**, por ejemplo, una aplicación muy deseable sería un **contrato inteligente** (contrato de cobertura en este caso) que cubriera contra la **volatilidad del ether** (u otra criptomoneda) con respecto al **dólar estadounidense,** pero para poder hacer esto se requiere que el contrato sepa cuál es el **valor entre ETH/ USD**. La forma más simple de hacer esto es a través de un contrato de **fuente de datos** mantenido por una parte específica designada (**NASDAQ**) de manera que esa parte tiene la habilidad de actualizar el contrato cuando es necesario y proporcionando un interfaz que permite a otros contratos enviar un mensaje a éste y obtener una respuesta de vuelta con el precio.

Sistemas de Identificación y Reputación. El contrato es muy simple; es una **base de datos** dentro de la **red Ethereum** donde se **puede agregar**, pero **no modificar** o **eliminar**. Cualquier **persona** puede registrar un **nombre con algún valor** y ese **registro queda para siempre**. Un contrato más sofisticado de registro de nombres también podría tener **cláusulas de función** que permitirían que otros contratos pudieran realizar consultas, así como un mecanismo para el **propietario** (es decir, el primero que registró) de un nombre que le habilitaría para **cambiar los datos o transferir la propiedad**. Incluso en la parte superior se podría agregar funcionalidad asociadas a la **reputación o a confianza en la web**.

Almacenamiento Descentralizado de Ficheros. En los últimos años, han surgido una serie de nuevas empresas de almacenamiento de archivos en línea muy populares (siendo Dropbox la más prominente de todas) y que buscan permitir a los usuarios

subir una copia de seguridad de su disco duro a la que puede acceder a cambio de una cuota mensual. Sin embargo, en este momento el mercado de almacenamiento de archivos es a veces relativamente ineficiente y una mirada superficial a diferentes soluciones existentes muestra que, sobre todo en el "valle inquietante" que se situaría en el nivel de los 20 a los 200GB y en donde no entran en juego ni las opciones gratuitas ni los descuentos de nivel empresarial, los precios mensuales son más de lo que estaríamos pagando por el disco duro en un solo mes. Los contratos de Ethereum pueden permitir el desarrollo de un ecosistema de almacenamiento de archivos descentralizado, donde los usuarios individuales pueden ganar pequeñas cantidades de dinero por el alquiler de sus propios discos duros y el espacio no utilizado, lo que se puede utilizar para impulsar costos de almacenamiento de archivos más bajos.

Organizaciones Autónomas Descentralizadas. El concepto general de una "organización autónoma descentralizada" es el de una entidad virtual que tiene un cierto conjunto de socios o accionistas que, tal vez con una mayoría del 67%, tienen el derecho de gastar los fondos de la entidad y modificar su código. Los miembros decidirán colectivamente sobre cómo la organización debe asignar sus fondos. Los métodos para la asignación de los fondos de una DAO podrían variar y ser del tipo de: recompensas, sueldos o incluso mecanismos aún más exóticos, tales como una moneda interna para recompensar el trabajo. Esto esencialmente replica las trampas legales de una empresa tradicional o sin ánimo de lucro pero utilizando la tecnología criptográfica de la cadena de bloques para su ejecución. Hasta ahora la mayor parte de la charla alrededor de las DAO's ha sido sobre un modelo "capitalista" de "corporación autónoma descentralizada" (DAC) con acciones negociables y accionistas que reciben dividendos; o también sobre otra alternativa descrita como "comunidad autónoma descentralizada", donde todos los miembros tienen una participación igual en la toma de decisiones y que requiere que al menos el 67% de los miembros existentes estén de acuerdo para añadir o eliminar miembros. El requisito para que una persona pueda estar afiliada necesitaría entonces ser impuesta colectivamente por el grupo.

Otras Aplicaciones

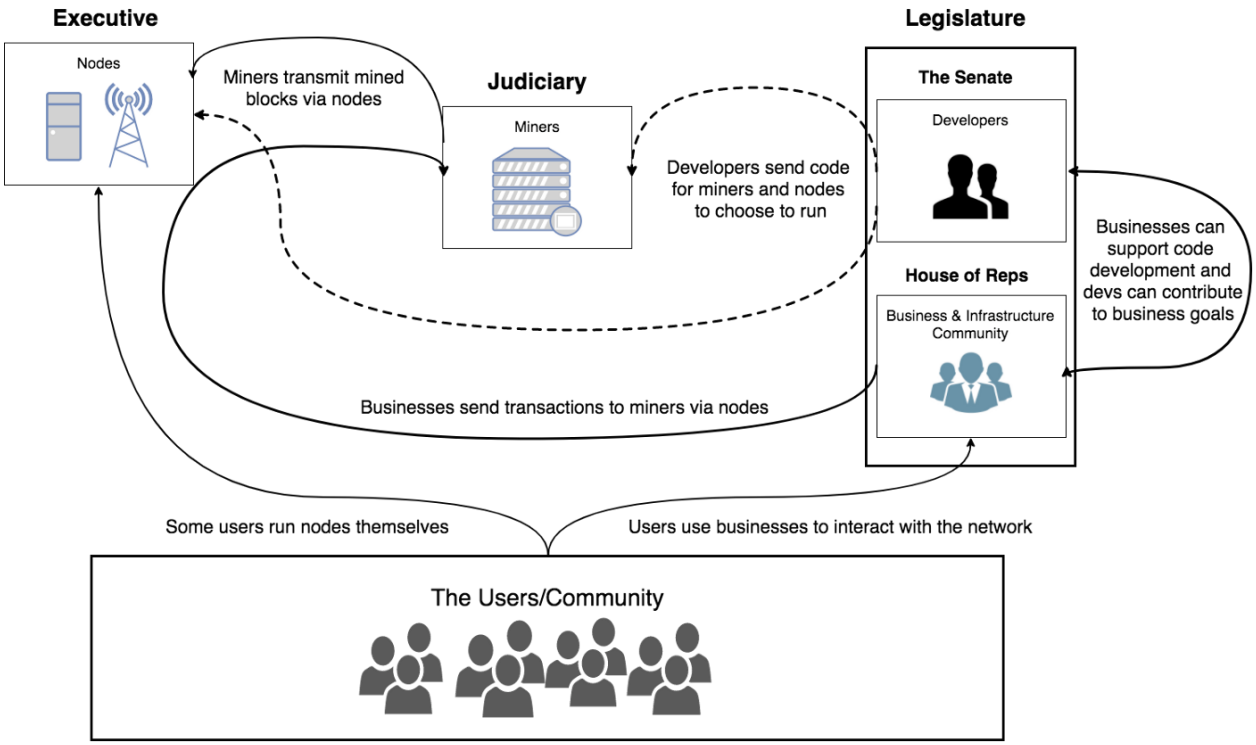
1. **Carteras de Ahorro.**
2. **Seguros para cosechas.**
3. **Fuentes de datos descentralizadas.**
4. **Custodia multi-firma inteligente.**
5. **Computación en la nube.**
6. **Sistemas de juegos y apuestas peer to peer.**

7. **Predicción de Mercados.**
8. **Mercados Descentralizados On-chain.**

Ethereum en la Administración Pública

La plataforma **Ethereum** a través de la implementación de los **contratos inteligentes** tiene una gama amplia de usos. **Sistemas de votación, registro de propiedades, registros catastrales, registros de nacimiento, sistema de patentamientos de vehículos, registros de escolaridad, pagos impositivos automatizados, sistema de licitaciones públicas, contratación de personal y nómina del personal** entre otras.

La plataforma **Ethereum**, al ser una **red de base de datos distribuida** permite promocionar la **transparencia de la gestión administrativa**, logrando acotar los altos niveles de corrupción. También al ser una **base de datos pública** permite una mejor gestión de la misma y obtener mejores estadísticas. A partir del **big data** es posible reconocer **patrones de conductas**, lo que posibilita una mejor **planificación de la gestión administrativa y ciudadana**.



Aplicación full stack de un sistema de votación en Ethereum

Está aplicación muestra el **código** utilizado para desarrollar una **aplicación descentralizada (DAPP)** de **votación.** La **prueba de concepto** es un **sistema de voto simple**.

Utilizaremos una **aplicación** en el **blockchain real**. **Ethereum** tiene dos **blockchain públicas**:

1. **Testnet** (también llamado **Ropsten**): es una blockchain. Consideremos a **Testnet** como un control de calidad o un servidor de ensayo, que se utiliza para propósitos de prueba. Todo el valor **ether** se utiliza en esta red es **falso**.
2. **MainNet** (también llamado **Homestead**): Es la **cadena blockchain**, que se utiliza para efectuar una **transacción de bienes**. Hay un **valor real** que se utiliza en esta red: **ether**.

Los objetivos son:

1. **Instalar Geth**. El **software del cliente** utilizado para descargar el **blockchain** y **ejecutar el nodo Ethereum en su máquina local**.
2. **Instalar el framework Ethereum DAPP llamado Truffle**. Que se utilizará para la **elaboración y la implementación de nuestro contrato**.
3. Hacer pequeños cambios a nuestra **solicitud de votación** para que funcione usando **Truffle**.
4. Compilar y desplegar el **contrato** en la **Testnet (Ropsten)**.
5. Interactuar con el contrato a través de la consola de **Truffle** y luego a través de una **página web**.

1. Instalar Geth y sincronizar el blockchain

He instalado y probado el stack en Ubuntu y MacOS. La instalación es bastante sencilla. En Mac:

En Mac:

```
@votacion:~$ brew tap ethereum/ethereum
@votacion:~$ brew install ethereum
```

En Ubuntu:

```
@votacion:~$ sudo apt-get install software-properties-common
@votacion:~$ sudo add-apt-repository -y ppa:ethereum/ethereum
@votacion:~$ sudo apt-get update
@votacion:~$ sudo apt-get install ethereum
```

Instrucciones de instalación para varias plataformas

Una vez que hayamos instalado **geth**, ejecutamos el siguiente comando en la consola de línea de comandos:

```
@votacion: ~ $ geth --testnet --syncmode "fast" --rpc --rpcapi db, ETH, red, web3, personal --cache = 1024 --rpcport 8545 --rpcaddr 127.0.0.1 --rpccorsdomain "*" --bootnodes "eNode://20c9ad97c081d63397d7b685a412227a40e23c8bdc6688c6f37e97cfbc22d2b4d1db1510d8f61e6a8866ad7f0e17c02b14182d37ea7c3c8b9c2683aeb6b733a1@52.169.14.227: 30303, eNodo: // 6ce05930c72abc632c58e2e4324f7c7ea478cec0ed4fa2528982cf34483094e9cbc9216e7aa349691242576d552a2a56aaeae426c5303ded677ce455ba1acd9d@13.84.180.240: 30303"
```

Esto iniciará el **nodo Ethereum**, se conecta a otros **nodos pares** e inicia la descarga del **blockchain**. El tiempo que tarda en **descargar el blockchain** depende de varios factores como la **velocidad de su conexión a Internet**, **memoria RAM** del equipo, t**ipo de disco duro**, etc. Lleva de **10-15 minutos** en una máquina que tiene conexión de **8 GB de RAM y 50 Mbps**.

En la consola en la que estamos ejecutando **Geth**, veremos la salida como se muestra a continuación. Busque el número de bloque que está en **negrita**. Cuando su b**lockchain está totalmente sincronizado**, el número de bloque que veremos estará muy cercano al número de bloque de esta página: Compara aquí.

```
I0130 22:18:15.116332 core/blockchain.go:1064] imported 32 blocks, 49 txs ( 6.256 Mg) in 185.716ms (33.688 Mg/s). #445097 [e1199364… / bce20913…]

I0130 22:18:20.267142 core/blockchain.go:1064] imported 1 blocks, 1 txs ( 0.239 Mg) in  11.379ms (20.963 Mg/s). #445097 [b4d77c46…]

I0130 22:18:21.059414 core/blockchain.go:1064] imported 1 blocks, 0 txs ( 0.000 Mg) in   7.807ms ( 0.000 Mg/s). #445098 [f990e694…]
```

```
I0130 22:18:34.367485 core/blockchain.go:1064] imported 1 blocks, 0 txs ( 0.000 Mg) in   4.599ms ( 0.000 Mg/s). #445099 [86b4f29a…]

I0130 22:18:42.953523 core/blockchain.go:1064] imported 1 blocks, 2 txs ( 0.294 Mg) in   9.149ms (32.136 Mg/s). #445100 [3572f223…]
```

2. Instalar el Framework Truffle

Instalar **Truffle** usando **npm**. La **versión de Truffle** utilizada es la **3.1.1**

```
@votacion:~$ npn install -g truffle
```

***** Dependiendo de la configuración del sistema, puede que tenga que añadir **sudo** al principio.*

3. Establecer el contrato de votación

El primer paso es configurar el proyecto de **truffle**:

```
@votacion:~$ mkdir voting
@votacion:~$ cd voting
@votacion:~/voting:~$ npm install -g webpack
@votacion:~/voting$ truffle init webpack
@votacion:~/voting$ ls
README.md               contracts         node_modules          test
webpack.config.js       truffle.js
app                     migrations        package.json
@votacion:~/voting$ ls app/index.html
javascripts  stylesheets
@votacion:~/voting$ ls contracts/
ConvertLib.sol          MetaCoin.sol    Migrations.sol
@votacion:~/voting$ ls migrations/
1_initial_migration.js    2_deploy_contracts.js
```

Como podemos observar arriba, **truffle** crea los archivos y directorios necesarios para ejecutar un **DAPP full stack**. **Truffle** también crea una aplicación de ejemplo para que podamos comenzar (no la vamos a utilizar en este caso). Podemos eliminar los archivos **ConvertLib.sol** y **MetaCoin.sol** en el **directorio de contracts** para este proyecto.

Es importante entender el contenido del directorio **migrations.** Estos archivos de **migrations** se utilizan para implementar los contratos en la **cadena blockchain**. El

archivo **1_initial_migration.js** en el directorio **migrations** despliega un contrato llamado **Migrations** en la **cadena blockchain** y se utiliza para almacenar el último contrato que haya implementado. Cada vez que ejecutamos **migrations**, **truffle** consulta a la cadena blockchain para obtener el último contrato que se ha desplegado y luego despliega un contrato que no se ha desplegado aún. A continuación, actualiza el campo **last_completed_migration** en el contrato de **Migrations** para indicar el último contrato desplegado. Simplemente, pensamos en ella como una tabla de base de datos llamada **Migration** con una columna llamada **last_completed_migration** que siempre se mantiene actualizada. Más detalles en la página de documentación de Truffle.

En primer lugar, escribimos en el lenguaje Solidity el siguiente código y llamamos al archivo **Voting.sol** y lo guardamos en el directorio **contracts**.

```
pragma solidity ^0.4.11;
// We have to specify what version of compiler this code will compile with

contract Voting {
  /* mapping field below is equivalent to an associative array or hash.
  The key of the mapping is candidate name stored as type bytes32 and value is
  an unsigned integer to store the vote count
  */

  mapping (bytes32 => uint8) public votesReceived;

  /* Solidity doesn't let you pass in an array of strings in the constructor (yet).
  We will use an array of bytes32 instead to store the list of candidates
  */

  bytes32[] public candidateList;

  /* This is the constructor which will be called once when you
  deploy the contract to the blockchain. When we deploy the contract,
  we will pass an array of candidates who will be contesting in the election
  */
  function Voting(bytes32[] candidateNames) {
    candidateList = candidateNames;
  }

  // This function returns the total votes a candidate has received so far
  function totalVotesFor(bytes32 candidate) returns (uint8) {
    if (validCandidate(candidate) == false) throw;
    return votesReceived[candidate];
```

```
  }

  // This function increments the vote count for the specified candidate. This
  // is equivalent to casting a vote
  function voteForCandidate(bytes32 candidate) {
    if (validCandidate(candidate) == false) throw;
    votesReceived[candidate] += 1;
  }

  function validCandidate(bytes32 candidate) returns (bool) {
    for(uint i = 0; i < candidateList.length; i++) {
      if (candidateList[i] == candidate) {
        return true;
      }
    }
    return false;
  }
}
```

```
@votacion:~/voting$ ls contracts/
Migrations.sol  Voting.sol
```

A continuación, **reemplazar** el contenido del archivo **2_deploy_contracts.js** ubicado en el directorio **migrations** de la siguiente forma:

```
var Voting = artifacts.require("./Voting.sol");
module.exports = function(deployer) {
 deployer.deploy(Voting, ['Rama', 'Nick', 'Jose'], {gas: 6700000});
};
/* Como observamos arriba, el desarrollador espera que el primer argumento sea el nombre del contrato seguido por los argumentos del constructor. En nuestro caso, sólo hay un argumento que es un conjunto de candidatos. El tercer argumento es un hash donde especificamos el gas necesario para implementar nuestro código. La cantidad de gas varía en función del tamaño de su contrato. Para el contrato de votación, 290000 fue suficiente.
*/
```

También podemos establecer el valor de **gas** como una configuración global en **truffle.js**. Agregamos la opción de **gas** a continuación, de modo que en el futuro si

nos olvidamos de configurar **gas** en un archivo de migración específico, usará de forma predeterminada el **valor global**.

```
require('babel-register')
module.exports = {
 networks: {
   development: {
     host: 'localhost',
     port: 8545,
     network_id: '*',
     gas: 470000
   }
 }
}
```

Reemplazamos el contenido de **app/javascript/app.js** con el contenido de abajo.

```
// Import the page's CSS. Webpack will know what to do with it.
import "../stylesheets/app.css";

// Import libraries we need.
import { default as Web3} from 'web3';
import { default as contract } from 'truffle-contract'

/*
 * When you compile and deploy your Voting contract,
 * truffle stores the abi and deployed address in a json
 * file in the build directory. We will use this information
 * to setup a Voting abstraction. We will use this abstraction
 * later to create an instance of the Voting contract.
 * Compare this against the index.js from our previous tutorial to see the difference
 *
https://gist.github.com/maheshmurthy/f6e96d6b3fff4cd4fa7f892de8a1a1b4#file-index-js
 */

import voting_artifacts from '../../build/contracts/Voting.json'

var Voting = contract(voting_artifacts);

let candidates = {"Rama": "candidate-1", "Nick": "candidate-2", "Jose": "candidate-3"}
```

```
window.voteForCandidate = function(candidate) {
  let candidateName = $("#candidate").val();
  try {
    $("#msg").html("Vote has been submitted. The vote count will increment as soon
as the vote is recorded on the blockchain. Please wait.")
    $("#candidate").val("");

    /* Voting.deployed() returns an instance of the contract. Every call
     * in Truffle returns a promise which is why we have used then()
     * everywhere we have a transaction call
     */
    Voting.deployed().then(function(contractInstance) {
      contractInstance.voteForCandidate(candidateName, {gas: 140000, from:
web3.eth.accounts[0]}).then(function() {
        let div_id = candidates[candidateName];
        return contractInstance.totalVotesFor.call(candidateName).then(function(v) {
          $("#" + div_id).html(v.toString());
          $("#msg").html("");
        });
      });
    });
  } catch (err) {
    console.log(err);
  }
}

$( document ).ready(function() {
  if (typeof web3 !== 'undefined') {
    console.warn("Using web3 detected from external source like Metamask")
    // Use Mist/MetaMask's provider
    window.web3 = new Web3(web3.currentProvider);
  } else {
    console.warn("No web3 detected. Falling back to http://localhost:8545. You should
remove this fallback when you deploy live, as it's inherently insecure. Consider
switching to Metamask for development. More info here:
http://truffleframework.com/tutorials/truffle-and-metamask");
    // fallback - use your fallback strategy (local node / hosted node + in-dapp id mgmt
/ fail)
    window.web3 = new Web3(new
Web3.providers.HttpProvider("http://localhost:8545"));
  }

  Voting.setProvider(web3.currentProvider);
  let candidateNames = Object.keys(candidates);
```

```
  for (var i = 0; i < candidateNames.length; i++) {
    let name = candidateNames[i];
    Voting.deployed().then(function(contractInstance) {
      contractInstance.totalVotesFor.call(name).then(function(v) {
        $("#" + candidates[name]).html(v.toString());
      });
    })
  }
});
```

Reemplazamos el contenido de **app/index.html** con el siguiente código:

```
<!DOCTYPE html>
<html>
<head>
  <title>Hello World DApp</title>
  <link href='https://fonts.googleapis.com/css?family=Open+Sans:400,700'
rel='stylesheet' type='text/css'>
  <link
href='https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css'
rel='stylesheet' type='text/css'>
</head>
<body class="container">
  <h1>A Simple Hello World Voting Application</h1>
  <div id="address"></div>
  <div class="table-responsive">
    <table class="table table-bordered">
      <thead>
        <tr>
          <th>Candidate</th>
          <th>Votes</th>
        </tr>
      </thead>
      <tbody>
        <tr>
          <td>Rama</td>
          <td id="candidate-1"></td>
        </tr>
        <tr>
          <td>Nick</td>
          <td id="candidate-2"></td>
        </tr>
        <tr>
          <td>Jose</td>
```

```
        <td id="candidate-3"></td>
       </tr>
     </tbody>
   </table>
   <div id="msg"></div>
  </div>
  <input type="text" id="candidate" />
  <a href="#" onclick="voteForCandidate()" class="btn btn-primary">Vote</a>
</body>
<script
src="https://cdn.rawgit.com/ethereum/web3.js/develop/dist/web3.js"></script>
<script src="https://code.jquery.com/jquery-3.1.1.slim.min.js"></script>
<script src="app.js"></script>
</html>
```

4. Implementar el contrato para la red de prueba Ropsten

Antes de que podamos desplegar el c**ontrato**, vamos a necesitar una **cuenta y ether**. Al utilizar **testnet y MainNet** tenemos que crear una **cuenta** y añadir **ether's** nosotros mismos. En el terminal de línea de comandos, escribimos:

```
@votacion:~/voting$ truffle console

truffle(default)> web3.personal.newAccount('verystrongpassword')
'0x95a94979d86d9c32d1d2ab5ace2dcc8d1b446fa1'

truffle(default)>
web3.eth.getBalance('0x95a94979d86d9c32d1d2ab5ace2dcc8d1b446fa1')
{ [String: '0'] s: 1, e: 0, c: [ 0 ] }

truffle(default)>
web3.personal.unlockAccount('0x95a94979d86d9c32d1d2ab5ace2dcc8d1b446fa1',
'verystrongpassword', 15000)

// Reemplazamos "verystrongpassword" con una buena contraseña segura.
// La cuenta está bloqueada por defecto, debemos desbloquearla antes de usar la
cuenta para el despliegue y la interacción con el blockchain.
```

En el post anterior, comenzamos un nodo en la consola inicializando el **objeto web3**. Cuando ejecutamos en la consola **truffle**, todo lo que hace es obtener un **objeto web3** listo para usar. Ahora tenemos una cuenta con la dirección

“0x95a94979d86d9c32d1d2ab5ace2dcc8d1b446fa1” (que tendrá una dirección diferente en vuestro caso) y el saldo será 0.

Podemos **“minar”** algunos **ether** mediante la ejecución del **nodo geth** y pasando una opción adicional **--mine**. La opción más fácil por ahora es conseguir ether del **faucet** o **ping me** y te enviará algunos. Verificamos **web3.eth.getBalance** de nuevo para asegurarnos que tenemos **ether**.

Ahora que tenemos algunos **ether**, podemos continuar compilando y desplegando el contrato a la cadena blockchain. Observamos el comando para ejecutar y la salida que muestra si todo va bien.

**** No se olvide de desbloquear la cuenta antes de implementar el contrato.***

```
@votacion:~/voting$ truffle migrate
Compiling Migrations.sol...
Compiling Voting.sol...
Writing artifacts to ./build/contracts
Running migration: 1_initial_migration.js
Deploying Migrations...
Migrations: 0x3cee101c94f8a06d549334372181bc5a7b3a8bee
Saving successful migration to network...
Saving artifacts...
Running migration: 2_deploy_contracts.js
Deploying Voting...
Voting: 0xd24a32f0ee12f5e9d233a2ebab5a53d4d4986203
Saving successful migration to network...
Saving artifacts...
mahesh@projectblockchain:~/voting$
```

Puede tomar unos **70-80 segundos para desplegar los contratos**.

5. La interacción con el contrato de votación

Si logramos implementar con éxito el contrato, ahora deberíamos ser capaces de buscar el recuento de votos y votar a través de la consola **truffle**.

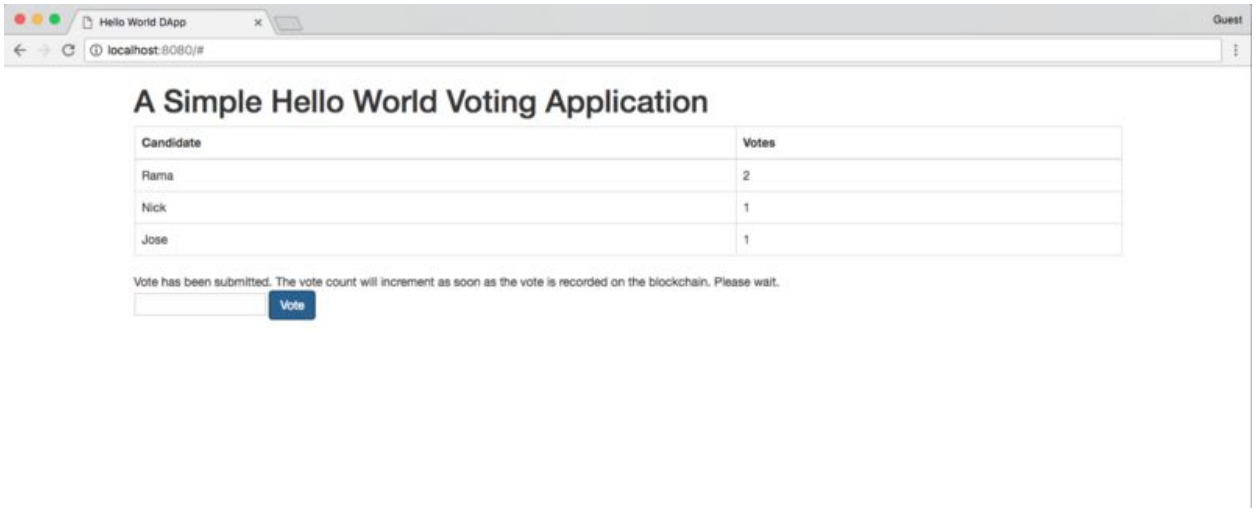
```
@votacion:~/voting$ truffle console

truffle(default)> Voting.deployed().then(function(contractInstance)
{contractInstance.voteForCandidate('Rama').then(function(v) {console.log(v)})})
```

```
// After a few seconds, you should see a transaction receipt like this:

receipt:

{ blockHash:
'0x7229f668db0ac335cdd0c4c86e0394a35dd471a1095b8fafb52ebd7671433156',

blockNumber: 469628,

contractAddress: null,
....
....

truffle(default)> Voting.deployed().then(function(contractInstance)
{contractInstance.totalVotesFor.call('Rama').then(function(v) {console.log(v)})})

{ [String: '1'] s: 1, e: 0, c: [ 1] }
```

Al llegar a este punto el **contrato es real y funcional.** Ahora, seguimos adelante y arrancamos el **servidor**:

```
@votacion:~/voting$ npm run dev
```

Deberíamos ver la **página de votación** en el **localhost: 8080**, **poder votar y ver los recuentos de votos de todos los candidatos**. Dado que se trata de una **verdadera blockchain**, cada escritura al **blockchain** (**voteForCandidate**) tomará unos segundos (Los **mineros** tienen que incluir su **transacción en un bloque** y el **bloque a la cadena blockchain**).



Si vemos esta página y somos capaces de votar entonces hemos construido una **aplicación Ethereum** en la red de prueba pública, **¡felicitaciones!**

Puesto que todas las transacciones son públicas, podemos verlas aquí: https://testnet.etherscan.io/. Solo tenemos que introducir la **dirección de la cuenta** y mostrará todas las **transacciones con marca de tiempo**.

Conclusiones

Esta prueba de concepto demuestra lo sencillo que crear un "smart contract" para un sistema de votación. Las ventajas son muy importantes y evidentes:

1. Inmutabilidad (no puede ser alterada ninguna transacción y bloque).
2. Al ser una base de datos pública todo el mundo puede acceder a sus datos.
3. Por ser una base de datos distribuida nadie nadie puede controlar la misma.
4. Se pueden desarrollar diferentes tipos de contratos sobre estos datos, para su procesamiento y análisis.

Podemos concluir que el uso de una cadena Blockchain, en este caso la **plataforma Ethereum**, genera una serie de ventajas vitales. Dichas ventajas permiten una gestión **democrática** y **eficiente** de los **datos**, además de una **excelente seguridad brindada** por los **sistemas criptográficos** utilizados en una **cadena blockchain**. Tampoco debemos dejar de considerar la gran cantidad de capas de intermediación eliminadas, lo cual permite reducir costos y aumentar la eficiencia del sistema.

@dmery