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**.

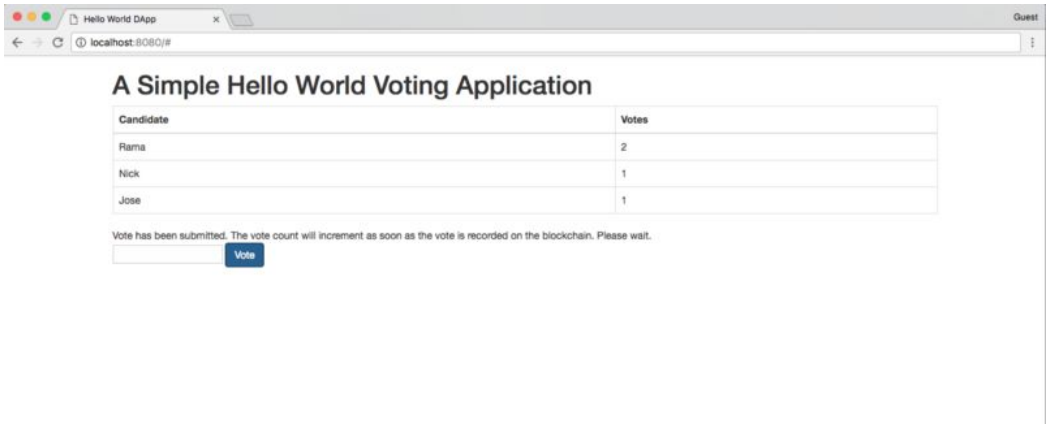
```
@votacion:~/voting$ truffle console

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