Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación

Desarrollo de una aplicación basada en la tecnología de Cadena de Bloques sobre la plataforma Ethereum que permite recolectar y almacenar los datos de las transacciones en una base de datos NoSQL.

Trabajo especial de grado
presentado ante la Ilustre
Universidad Central de Venezuela
Por el bachiller:
Carlos Pires
Tutor:
Prof. Jesús Lares
Caracas, Mayo de 2018

Acta

Quienes suscriben, miembros del jurado designado por el Consejo de la Escuela de Computación, para examinar el Trabajo Especial de Grado titulado Desarrollo de una aplicación basada en la tecnología de Cadena de Bloques sobre la plataforma Ethereum que permite recolectar y almacenar los datos de las transacciones en una base de datos NoSQL presentado por El Br. Carlos David Pires Días (C.I. V-22667768) a los fines de optar al título de Licenciado en Computación, dejamos constancia de lo siguiente:

Leído como fue dicho trabajo, por cada uno de los miembros del jurado, se fijó el día 24 de mayo, a las 11 horas de la mañana, para que el (los) autor(es) lo defendiera(n) en forma pública, lo que este (esta/estos) hizo (hicieron) en el Centro de Computación, mediante una presentación oral de su contenido, luego de lo cual respondieron a las preguntas formuladas.

Finalizada la defensa pública del Trabajo Especial de Grado, el jurado decidió aprobarlo.
Es de aclarar que el Profesor Jesús Lares se encuentra fuera del país, sin embargo estuvo presente en la presentación, vía remota a través de una videollamada. Por esta razón, el Profesor Robinson Rivas, Director de la Escuela de Computación, firma el presente documento en su lugar

En fe de lo cual se levanta la presente Acta, en Caracas el día 24 de mayo de 2018.

por: RRR.

Prof(a). Jesús Lares (Tutor)

REPUBLICA DE VENEZUELA
Facultad de Ciencias
Escuela de Computación
Universidad Central - Caracas

Profa. Ospina Mercy (Jurado)

Prof(a). Núñez Haydemar (Jurado)

Agradecimientos

Principalmente a mi madre, por hacerme la persona que hoy soy, y por brindarme el apoyo para ser la persona que quiero ser.

A mi hermano, que a pesar de la distancia, me inspiró con su valentía y perseverancia.

A Ezequiel, el amigo verdadero.

Un agradecimiento muy especial a mi tutor Jesús Lares, por ser un docente ejemplar, líder atento, responsable, enemigo de la mediocridad y como dice él, un 'hermano contemporáneo'.

Mucha gratitud hacia Antonio Nardi, Horacio Coll y Rafael Olivares por ser profesionales de calidad, además de colegas de trabajo y buenos amigos que me brindaron la oportunidad de formarme como profesional y creer en mi trabajo.

A la Universidad Central de Venezuela, por la tarea titánica de educar, en contra de todas las adversidades.

Resumen

Título: Desarrollo de una aplicación basada en la tecnología de Cadena de Bloques sobre la plataforma Ethereum que permite recolectar y almacenar los datos de las transacciones sobre almacenes de datos NoSQL.

Autor: Carlos Pires.

Tutor: Jesús Lares.

El presente TEG expone las bondades que brindan las nuevas tecnologías de cadena de bloques y contratos inteligentes sobre plataformas diseñadas para albergar aplicaciones distribuidas, adicionalmente muestra el proceso de creación de una aplicación que se vale de una interfaz web para interactuar con dichas cadenas de bloques y finalmente, albergar dichas interacciones en almacenes de datos NoSQL con la finalidad de aplicar técnicas de analítica predictiva e inteligencia artificial que ayuden a detectar patrones de comportamiento u otro tipo de información de valor que almacenen las cadenas de bloques.

Palabras clave: Cadena de bloques, Contratos inteligentes, Transacciones, Ethereum, Aplicaciones distribuidas, NoSQL.

Algoritmos

6.1. Versión 0.1 de solución para el caso de uso 56
6.2. Código que genera el ABI y *Bytecode* de un contrato dado 62
6.3. Código que realiza pruebas automáticas de despliegue de un contrato y funcionalidades 63
6.4. Versión 0.2 de solución para el caso de uso 70
6.5. Código que realiza pruebas automáticas de despliegue de un contrato y funcionalidades. Versión 0.2 77
6.6. Versión 0.3 de solución para el caso de uso 83
6.7. Código que realiza pruebas automáticas de despliegue de un contrato y funcionalidades. Versión 0.3 89
6.8. Versión 1.0 de solución para el caso de uso 96
6.9. Módulo que genera una instancia de Web3 conectada a la red Rinkeby(web3.js) 104
6.10. Módulo que realiza el despligue del contrato en la red Rinkeby (deploy.js) . . 105
6.11. Módulo que interactúa con un contrato dada una dirección y su ABI (factory.js) 106
6.12. Módulo que interactúa con un contrato dada una dirección pasada por parámetro y su ABI (campaign.js) 107
6.13. Módulo que inicia en una red local la aplicación web(server.js) 109
6.14. Módulo que configura las rutas del servidor web(routes.js) 110
6.15. Ruta que sirve como página base de la aplicación web. Muestra la lista de campañas desplegadas (index.js) 110
6.16. Ruta que muestra el formulario para la creación de nuevas campañas (campaigns/new.js) 112
6.17. Ruta que muestra los detalles de una campaña dada una dirección particular (campaigns/show.js) 116
6.18. Módulo que maneja las contribuciones a una campaña en específico (Components/ContributionForm.js) 120
6.19. Módulo que maneja las solicitudes a una campaña en específico (campaigns/-requests/index.js) 123
6.20. Módulo que se encarga de construir cada una de las filas de la tabla de solicitudes (Components/RequestRow.js) 126
6.21. Módulo que se encarga de construir el formulario para crear una solicitud (pages/campaigns/requests/news.js) 132

6.22. Componente que hace de cabecera a todas las páginas del cliente web (Components/Header.js) . . . 135
6.23. Componente que hace contenedor a todas las páginas del cliente web (Components/Layout.js) . . . 136
6.24. Componente que se conecta con MongoDB (server/app.js) . . . 137
6.25. Ruta que asigna el módulo controlador que maneja las solicitudes (server/-routes/router.js) . . . 138
6.26. Modelo de una campaña para el almacenamiento en la base de datos (server/models/campaign.model.js) . . . 138
6.27. Modelo de una contribución para el almacenamiento en la base de datos (server/models/contribution.model.js) . . . 140
6.28. Modelo de una solicitud para el almacenamiento en la base de datos (server/-models/request.model.js) . . . 140
6.29. Modelo de una aprobación de una solicitud para el almacenamiento en la base de datos (server/models/approved.model.js) . . . 141
6.30. Modelo de un rechazo de una solicitud para el almacenamiento en la base de datos (server/models/rejected.model.js) . . . 142
6.31. Modelo de finalización de una solicitud para el almacenamiento en la base de datos (server/models/finalized.model.js) . . . 143
6.32. Módulo encargado de manejar las solicitudes que provengan del lado del cliente y almacenar en la base de datos utilizando los modelos definidos (server/-main.controller.js) . . . 144

Lista de Tablas

6-1. Especificaciones de la prueba para la versión 0.1 60
6-2. Especificaciones de la prueba automática para la versión 0.1 68
6-3. Especificaciones de hardware para pruebas en Mocha 68
6-4. Especificaciones de la prueba para la versión 0.2 75
6-5. Especificaciones de hardware para pruebas en Mocha. Ambiente MacOS . . . 76
6-6. Especificaciones de la prueba para la versión 0.3 89
6-7. Especificaciones de hardware para pruebas en Mocha, versión 0.3. Ambiente MacOS 95
6-8. Especificaciones de la prueba de aceptacón para interfaces gráficas. 148
6-9. Tabla para evaluación de interfaces gráficas. 148
6-10.Tabla para evaluación de la pantalla principal del cliente web. 150
6-11.Tabla para evaluación de la pantalla creación de una campaña. 152
6-12.Tabla para evaluación de botón de contribución para una campaña 154
6-13.Tabla para evaluación de detalles de una campaña. 156
6-14.Tabla para evaluación de creación de una solicitud. 158
6-15.Tabla para evaluación de lista de solicitudes. 161

Lista de Figuras

2-1. Representación gráfica de la arquitectura propuesta para el TEG. 7

3-1. Representación gráfica de una cadena de bloques.
Recuperado y traducido de: *https* : *//bitcoin.org/en/developer − guide* . . . 10
3-2. Representación simplificada de la estructura de datos que enlaza bloques marcados de tiempo. 13
3-3. Cadena de bloques y el árbol Merkle.
Recuperado y traducido de: *https* : *//bitcoin.org/bitcoin.pdf* 15
3-4. Representación gráfica de la cadena de bloques de Ethereum. 22
3-5. Ethereum y su cadena de bloques en la arquitectura planteada. 23
3-6. Representación gráfica de una transacción en Ethereum. 25
3-7. Ejemplo de costos de *GAS*. 26
3-8. Ejemplo de costo final de una transacción 26
3-9. Ejemplo minado de una transacción con dificultad 100 27

4-1. Capa intermedia planteada en la arquitectura. 36
4-2. Capa final planteada en la arquitectura. 37

5-1. Ciclo de vida del Proceso Ágil Unificado. 44
5-2. Iteraciones de liberación de versiones. 45

6-1. Gerente crea una instancia de contrato inteligente mediante un cliente web. . 48
6-2. Contribuyentes ingresan ether a la campaña mediante el cliente web. 49
6-3. Gerente se vale del cliente web para crear una solicitud de gasto. 50
6-4. Los clientes se valen del cliente web para aprobar o rechazar una solicitud de gasto de una campaña. 51
6-5. Cuando se haya alcanzado la cantidad necesaria de votos, el gerente puede finalizar la solicitud. 52
6-6. Todas las interacciones de parte de contribuyentes y el gerente con la campaña son registradas en una instancia de MongoDB. 53
6-7. Diagrama de caso de uso para el gerente de una campaña. 54
6-8. Diagrama de caso de uso para contribuyentes de una campaña. 55
6-9. IDE de Remix, compilador de Solidity en en navegador web. 61
6-10.Ejecución de pruebas automáticas mediante el uso de la biblioteca Mocha. . 69

6-11.Ejecución de pruebas automáticas mediante el uso de la biblioteca Mocha para la versioón 0.2 de la solución. 82
6-12.Ejecución de pruebas automáticas mediante el uso de la biblioteca Mocha para la versioón 0.3 de la solución.. 96
6-13.Despliegue del contrato en la red Rinkeby, retorna una dirección del contrato desplegado 106
6-14.Estructura del cliente web 108
6-15.Pantalla principal del cliente web. 149
6-16.Pantalla para la creación de una campaña. 151
6-17.Botón de contribución para una campaña. 153
6-18.Detalles de una campaña. 155
6-19.Creación de una solicitud. 157
6-20.Lista de solicitudes, una lista para aprobar, una lista para ser rechazada, dos en vigencia. 160
6-21.Lista de solicitudes, una expirada, una finalizada. 162
6-22.Error al dejar campos vacíos en la creación de una campaña. 162
6-23.Error al insertar caracteres al monto de contribución. 163

Contenido

1 Capítulo 1 -Introducción **2**

2 Capítulo 2 - Planteamiento del problema **4**

2.1 Planteamiento del problema 4
2.2 Justificación 5
2.3 Objetivo general: 5
2.4 Objetivos específicos: 6
2.5 Arquitectura Propuesta para TEG 6
2.6 Alcance del TEG 8

3 Capítulo 3 - Marco teórico **9**

3.1 Cadena de bloques (Blockchain) 9
3.1.1 Definición 9
3.2 Propiedades deseables 10
3.3 Antecedentes académicos 11
3.4 Bitcoin Blockchain 12
3.5 Árboles Merkle 12
3.6 Prueba de trabajo (Proof-Of-Work) 16
3.7 Casos de uso de las cadenas de bloques 16
3.7.0.1 Criptomonedas 16
3.7.0.2 Servicios financieros 17
3.7.0.3 Servicios de pólizas de seguros 17
3.7.0.4 Servicios Gubernamentales 18
3.7.0.5 Cadena de Suministros 18
3.7.0.6 Salud Púbica/Privada 18
3.7.0.7 Internet de las cosas (Internet of things) 19
3.8 Contratos inteligentes y Ethereum 19
3.8.1 Aplicaciones distribuidas (DAPP) 19
3.8.1.1 El nacimiento de las DAPPs 19
3.8.1.2 Definición de DAPP 19
3.8.1.3 Bitcoin como DAPP 20
3.8.1.4 Clasificación de las DAPP's 20
3.8.2 Ethereum 21
3.8.3 Ethereum Blockchain 21

3.8.4 Máquina Virtual de Ethereum . . . 22
3.8.5 Tipos de cuentas en Ethereum . . . 23
3.8.6 Transacciones . . . 24
3.8.7 Gas - Wei . . . 25
3.8.8 Minado . . . 26
3.8.9 Tiempo de minado y dificultad del bloque . . . 27
3.8.10 Ether . . . 27
3.8.11 Contratos Inteligentes(Smart Contracts) . . . 28
3.8.12 Contratos Inteligentes: Casos de uso . . . 29
3.8.12.1 Contratos inteligentes para Seguridad Financiera . . . 29
3.8.12.2 Contratos inteligentes para Procesos Derivados . . . 29
3.8.12.3 Contratos inteligentes para la Captación de Datos . . . 30
3.8.12.4 Contratos inteligentes para Hipotecas . . . 30
3.8.12.5 Contratos inteligentes en el Ámbito de Seguros . . . 30
3.8.12.6 Contratos inteligentes en el Ámbito Médico . . . 30
3.8.12.7 Contratos inteligentes junto a Internet de las cosas . . . 31

4 Capítulo 4 - Tecnologías utilizadas 32
4.1 Lenguajes de Programación . . . 32
4.1.1 JavaScript . . . 32
4.1.2 Solidity . . . 32
4.2 Remix . . . 33
4.3 Infura . . . 33
4.4 MetaMask . . . 33
4.5 NodeJS . . . 33
4.5.1 Web3 . . . 33
4.5.2 SolC . . . 34
4.5.3 Ganache . . . 34
4.5.4 Mocha . . . 34
4.6 React . . . 34
4.6.1 NextJS . . . 34
4.6.2 Semantic UI React . . . 34
4.7 Justificación de tecnologías . . . 35
4.7.1 MetaMask . . . 35
4.7.2 NodeJS + Web3 . . . 35
4.7.3 React + NextJS + Semantic UI React . . . 35
4.8 Almacenamiento . . . 38
4.9 Bases de Datos NoSQL . . . 38
4.9.1 Tipos de Bases de Datos NoSQL . . . 38
4.9.2 MongoDB . . . 39

4.9.3 Justificación de uso MongoDB . . . 39
4.10 Inteligencia Artificial . . . 39
4.10.1 Analítica Predictiva . . . 39
4.10.2 Aprendizaje Automático (Machine Learning) . . . 40
4.10.3 Aprendizaje profundo (Deep Learning) . . . 40
4.10.4 Minería de Datos (Data Mining) . . . 41
4.11 Alcance de la Investigación . . . 41

5 Capítulo 5 - Marco metodológico 42
5.1 Metodología de desarrollo y justificación . . . 42
5.2 Proceso Ágil Unificado . . . 42
5.2.1 Fases . . . 43
5.2.2 Disciplinas y actividades . . . 43
5.2.3 Iteraciones alrededor del tiempo . . . 44
5.2.4 Filosofía del Agile Unified Process . . . 46

6 Capítulo 6 - Marco aplicativo 47
6.1 Inicio (*Inception*) . . . 47
6.1.1 Propuesta de caso de uso para el TEG . . . 47
6.1.1.1 Solución para evitar malversación de bienes y lavado de dinero en recaudaciones masivas . . . 47
6.1.1.2 ¿Qué es una recaudación masiva? . . . 47
6.1.1.3 ¿Qué problemas presentan actualmente las recaudaciones masivas? . . . 47
6.1.1.4 ¿Cómo pueden Ethereum y los contratos inteligentes atacar estos problemas? . . . 47
6.1.1.5 Solución . . . 48
6.2 Elaboración (*Elaboration*) . . . 56
6.3 Construcción (*Construction*) . . . 56
6.3.1 Iteración 0: Acercamiento inicial . . . 56
6.3.1.1 Prueba 0.1 . . . 60
6.3.2 Iteración 1: Compilación y primera prueba automática . . . 62
6.3.3 Iteración 2: Segundo acercamiento, reestructuración de pruebas . . . 70
6.3.3.1 Prueba versión 0.2 . . . 75
6.3.4 Iteración 3: Rechazo de solicitudes y tasas . . . 82
6.3.4.1 Prueba versión 0.3 . . . 89
6.3.5 Iteración 4: Expiración de solicitudes . . . 96
6.3.6 Iteración 5: Construcción cliente web . . . 104
6.3.6.1 Modulo Web3 . . . 104
6.3.6.2 Despliegue del contrato en la red . . . 105

6.3.6.3 Referencias a contratos desplegados en la red 106
6.3.6.4 Cliente web 107
6.3.6.5 Rutas de la aplicación web 110
6.3.6.6 Creación de campañas 112
6.3.6.7 Visualizar los detalles de campañas 116
6.3.6.8 Contribuir a una campaña 120
6.3.6.9 Visualizar solicitudes 123
6.3.6.10 Creación de solicitudes 132
6.3.6.11 Elementos de interfaz 135
6.3.7 Iteración 6: Almacenamiento de interacciones 137
6.3.7.1 Servidor MongoDB 137
6.3.7.2 Modelos 138
6.4 Transición(*Transition*) 147
6.4.1 Iteración 0: Pruebas de interfaz de usuario 147
6.4.1.1 Crear campaña 151
6.4.1.2 Visualizar una campaña / Contribuir a una campaña 153
6.4.1.3 Crear una solicitud 157
6.4.1.4 Aprobar/Rechazar/Finalizar solicitudes 159
6.4.1.5 Validación y mensajes de error 162

7 Capítulo 7 - Conclusiones 164
7.0.1 Seleccionar Caso de Uso 164
7.0.2 Seleccionar una metodología de desarrollo 164
7.0.3 Preparar el ambiente de desarrollo de la plataforma Ethereum 164
7.0.4 Recopilación de datos 165
7.0.5 Definir almacén de datos 165
7.0.6 Implementar la aplicación sobre la base tecnológica de cadena de bloques y contratos inteligentes 165
7.0.7 Pruebas de funcionamiento 165
7.0.8 Pruesta en marcha 166
7.1 Contribuciones 166
7.2 Recomendaciones 166
7.3 Trabajos futuros 166

Bibliografía 167

1 Capítulo 1 -Introducción

El alza de la tecnología en esta época de la humanidad obliga a la sociedad a reinventarse, a medida que surgen nuevas necesidades y problemas que antes no existían, también surgen nuevas soluciones que pueden transformar todos los paradigmas que se consideraban tradicionales en nuevas prácticas más eficientes y que con un enfoque distinto, pueden hacerle frente a los problemas del futuro.

En una sociedad tan globalizada como la de hoy en día, se presentan nuevos desafíos en un abrir y cerrar de ojos, la información que se genera a diario necesita ser validada, y más importante aún, necesita ser preservada a medida que la población aumenta su magnitud de manera tan acelerada. Hace falta un nuevo enfoque que pueda brindarle confianza, seguridad y transparencia a los datos que se generan a diario.

Los sistemas centralizados cada vez se ven más afectados por sus problemas de escalabilidad al momento de enfrentarse con gigantescos volúmenes de usuarios. A la fecha, una transacción bancaria de grandes cantidades de dinero puede tardarse horas e incluso días en hacerse efectiva, por el ritmo que lleva la sociedad actual, tal cantidad de tiempo no es lo suficientemente corto como para saciar las necesidades tan específicas que requieren millones de empresas cada día.

Adicionalmente a eso, muchas de estas transacciones pasan por varias entidades antes de llegar a su destino final, y es bien conocido que por cada verificación que un tercero realiza a su cliente, todo tipo de impuestos son descontados del valor inicial, perdiéndose así gigantescas cantidades de dinero sólo en terceros que se encargan de hacer que el dinero llegue de un lugar a otro.

Lamentablemente también somos parte de una sociedad donde existe la corrupción y la malversación de fondos, existen grandes cantidades de empresas, fundaciones e incluso gobiernos que parecen desaparecer dinero y recursos sin que se vean reflejados en avances en sus proyectos o sociedades, dejando muchas dudas sobre quién y cómo se manejan tales fondos.

¿Por qué no existe un sistema directo, transparte, confiable y rápido? .

Las criptomonedas fueron los primeros protocolos en atacar los problemas antes mencionados,

las mismas brindan una plataforma segura, transparente y eficaz para realizar transacciones de bienes digitales.

La espina dorsal de las criptomonedas actuales es conocida como cadena de bloques(Blockchain, en inglés), y esta tecnología no se limita sólo al uso en criptomonedas, una cadena de bloques tiene tantos usos como la imaginación del ser humano puede darle.

La presente investigación realizará un análisis exahustivo con respecto al desarrollo de aplicaciones que interactúan con la tecnología de cadena de bloques mediante el uso de contratos inteligentes.

El capítulo II define el planteamiento del problema, justificación, el objetivo general, objetivos específicos y la arquitectura propuesta para el presente TEG.

A lo largo del capítulo III se encuentra el marco teórico, el cual define todos los conceptos teóricos y prácticos que sirven de referencia para el desarrollo planteado.

En el capítulo IV se describen las herramientas tecnológicas que se utilizarán durante el desarrollo junto con las justificaciones de uso de las mismas.

La metodología de desarrollo se describe a lo largo del capítulo V junto con su justificación y fases definidas.

El capítulo VI muestra las fases de la metodología de desarrollo aplicadas y sigue cuidadosamente el proceso del desarrollo de la herramienta, junto con pruebas y despligues en ambiente de desarrollo.

Finalmente, el capítulo VII corresponde a las conclusiones obtenidas durante todo el desarrollo del TEG.

2 Capítulo 2 - Planteamiento del problema

2.1. Planteamiento del problema

Las cadenas de bloques pueden aplicarse a todo tipo de ámbitos, desde una pequeña empresa, pasando por restaurantes, sellos discográficos, hospitales, empresas multinacionales y hasta administraciones gubernamentales. Básicamente puede aplicarse a cualquier escenario donde quiera llevarse un registro, desde el origen a través del tiempo, de un bien físico o virtual, como puede ser una imagen digital o un neumático de un auto. Con la tecnología que componen las cadenas de bloques, un paradigma descentralizado, transparente, confiable y directo se cierne sobre los enfoques tecnológicos actuales.

Los enfoques y nuevos paradigmas que plantean las cadenas de bloques y contratos inteligentes (Smart contracts, en inglés), prometen dar solución a una gran parte de los problemas que se plantean en los antecedentes de esta investigación, sin embargo, al ser una tecnología reciente surgen algunas incertidumbres que son de mucho interés en el ámbito mundial: ¿Es realmente la tecnología de cadena de bloques junto a los contratos inteligentes capaces de resolver tales problemáticas?, ¿Qué tan complejo es implementar una solución utilizando estas tecnologías?, ¿Qué tan efectivas son estas soluciones?, ¿En qué ambientes pueden ser implementadas?, ¿Cómo es posible realizar una recopilación datos dada una cadena de bloques?, ¿Es posible encontrar de patrones de comportamiento en esos datos?

Ethereum, una plataforma con su propia cadena de bloques y especializada en el alojamiento de contratos inteligentes con interactúan con ella, sirve como ambiente para el presente trabajo especial de grado, el cual plantea realizar, dado un caso de uso seleccionado, una herramienta que pueda explotar al máximo las ventajas que las cadenas de bloques y los contratos inteligentes pueden brindar para resolver distintas problemáticas (contempladas en la primera parte de este TEG), que requieren de soluciones eficientes y confiables. Adicionalmente la recopilación de datos que se agregan en las cadenas de bloques es una tarea fundamental en el desarrollo de la solución que se apoya en este tipo de enfoques jóvenes.

2.2. Justificación

Debido a la globalización creciente en la tecnología, es necesario estudiar el desarrollo de aplicaciones que comiencen a interactuar con estas nuevas tendencias, gracias a que las mismas pueden resolver los problemas que van surgiendo a medida que la sociedad evoluciona.

Los altos tiempos de espera de transacciones monetarias, las tediosas auditorías, la intermediación innecesaria de terceros en diferentes procesos de negociación y el desperdicio de espacio físico en documentos son algunas de las razones por las cuales se debe empezar a adoptar estos enfoques de manera global.

Con la adopción masiva de este tipo de tecnologías, viene siendo crucial encontrar maneras de recopilar datos que pueden de ser vital importancia para estudios futuros, ya que un vez que la forma de almacenar los datos está cambiando, también cambia la forma en la se obtienen esos datos debido a que se vuelven masivos muy rápidamente y la eficiencia es un factor fundamental en la recopilación.

La vialidad, es uno de los factores más importantes a la hora de pensar en cualquier solución tecnológica, el presente TEG también plantea evaluar qué tan viables son estos nuevos enfoques y desarrollos, qué tanta complejidad se despliega de este tipo de soluciones y finalmente, observar que tan eficientes y eficaces son con respecto al caso de uso seleccionado.

Adicionalmente, de cara a estudios futuros que se pueden realizar mediante técnicas como la analítica predictiva y la inteligencia artificial, junto con, la proyección que este tipo tecnologías tienen hacia el futuro debido a su transparencia y eficiencia, la tarea de recopilación de datos para ser analizados posteriormente y detectar patrones de comportamiento se vuelve fundamental para brindar una solución completa en todos los aspectos. La posibilidad de generar estos datos estructurados de una manera eficiente para que los expertos en el área de análisis de datos puedan realizar la tarea de analizar apropiadamente dichos datos es una característica innovadora y de vital importancia para este tipo de soluciones.

2.3. Objetivo general:

Desarrollar una aplicación basada en un caso de uso previamente seleccionado, sobre la tecnología de cadena de bloques (Blockchain) utilizando la plataforma Ethereum y sus contratos inteligentes y adicionalmente capturar las iteracciones de los usuarios con la cadena de bloques para ser almacenadas en una base de datos NoSQL.

2.4. Objetivos específicos:

1. Seleccionar Caso de Uso: Seleccionar una problemática existente a la cual sea viable la aplicación de las herramientas investigadas para dar con soluciones eficientes.

2. Seleccionar una metodología de desarrollo: Seleccionar la metodología de desarrollo que proponga una estructura de trabajo eficiente para desarrollar una solución eficaz al caso de uso planteado.

3. Preparar el ambiente de desarrollo de la plataforma Ethereum: Preparar las cuentas a utilizar en el ambiente Ethereum para solventar la problemática y proponer una implementación de contrato inteligente.

4. Recopilación de datos: Recopilar las interacciones de los usuarios finales con la cadena de bloques para brindar un conjunto de datos útiles para análisis futuros.

5. Seleccionar almacén de datos: Seleccionar un almacén de datos eficiente para que puedan aplicarse algoritmos de analítica predictiva e inteligencia artificial en el futuro.

6. Implementar la aplicación sobre la base tecnológica de cadena de bloques y contratos inteligentes: Implementar la aplicación de interacción con la cadena de bloques utilizando Solidity como herramienta desarrollo para resolver la problemática seleccionada.

7. Realizar pruebas de funcionamiento: Realizar pruebas a la aplicación en interacción con la Máquina Virtual de Ethereum en una red local, para verificar su funcionamiento y capacidad para resolver la problemática seleccionada.

8. Poner en marcha: Poner en marcha en un ambiente de producción la aplicación desarrollada para validar sus funcionalidades.

2.5. Arquitectura Propuesta para TEG

La presente arquitectura (Figura **2-1**) presenta una estructura de tres capas principales. La primera capa se encarga de la interacción directa con la cadena de bloques de Ethereum mediante contratos inteligentes, en esta capa ocurre la recopilación de datos a los cuales les serán aplicados los diferentes métodos análiticos y predictivos.

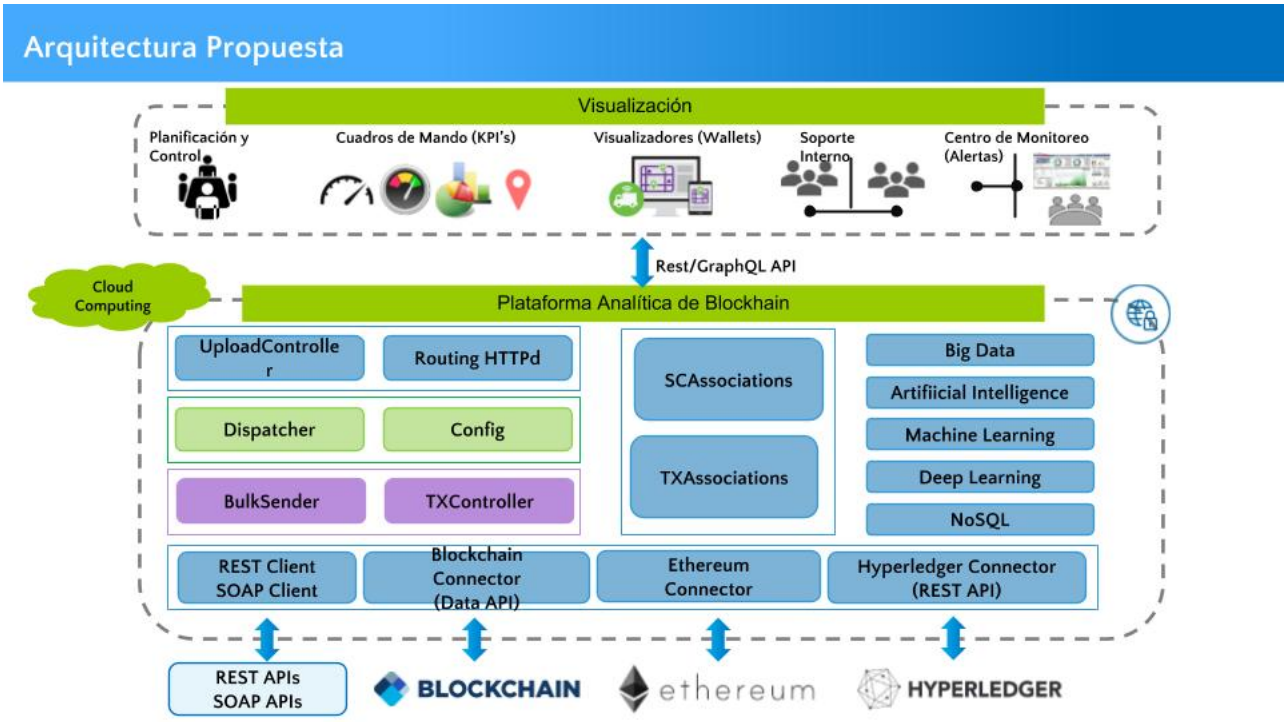

Figura 2-1: Representación gráfica de la arquitectura propuesta para el TEG.

En la segunda capa se realiza el preprocesamiento de los datos recopilados y los diferentes algoritmos de analítica predictiva e inteligecia artifial utilizando una plataforma Big Data. Es la capa intermedia entre la recopilación de los datos y la interfaz de usuario el cual se encarga de mostrar los resultados obtenidos.

Finalmente se encuentra la capa de visualización la cual mostrará los resultados obtenidos al usuario final desde la segunda capa. Esta tercera capa está provista de distintos métodos de visualización dependiendo del tipo de indicador que se especifique.

2.6. Alcance del TEG

El TEG pretende implementar sólo algunos módulos de la arquitectura planteada, los cuales serán definidos sobre la marcha a medida que se desarrolle el TEG, el resto de los módulos, debido a su amplio espectro de alcance, podrían ser implementados en otros TEG en el futuro.
La presente investigación se centra en la creación de módulos en la primera capa, con respecto al conector de Ethereum para la recopilación de datos, la definición y llenado del almacén de datos NoSQL contemplado en la segunda capa y posiblemente un elemento de presentación de la capa de visualización

3 Capítulo 3 - Marco teórico

3.1. Cadena de bloques (Blockchain)

3.1.1. Definición

Es un libro mayor digital, el cual lleva un registro permanente e incorruptible que facilita el proceso de registro de transacciones que involucran cualquier tipo de bienes (virtuales o no) en una red[35]. Ampliando un poco más este concepto, una cadena de bloques está compuesta por una serie de bloques enlazados cronológicamente y que contienen transacciones que no pueden modificarse una vez insertadas en la cadena. Esta herramienta se vale de la combinación de diferentes tecnologías tales como la criptografía, los Árboles Merkle, llaves asimétricas y firmas digitales para lograr la continuidad en el tiempo y la inmutabilidad.
La cadena comienza con un bloque génesis, con una marca de tiempo, de este bloque inicial nacen todos los demás bloques de la cadena, cuando se cree un nuevo bloque (el segundo bloque en este caso) este estará enlazado con el bloque génesis a través de un apuntador que tendra direccón al código SHA-256 del primer bloque, cuando un tercer bloque sea creado, se enlazará con el código SHA-256 del anterior y así sucesivamente (con sus respectivas marcas de tiempo). De esta forma, el protocolo se apoya en la criptografía para crear un sistema incorruptible, ya que, si alguien quisiera modificar algún bloque en el pasado de la cadena, esto modificaría su hash, lo cual ocasionaría que todos los demás bloques tuvieran apuntadores incorrectos, lo que dispararía un error inmediatamente en la red.
Los bloques pueden tener apuntadores a los códigos hash de las raíces de Árboles Merkle los cuales almacenan transacciones de una manera eficiente para poder tener un rápido acceso a ellas en caso de requerir alguna consulta, si bien esta implementación no es necesaria, se ha vuelto muy popular a través de diferentes cadenas de bloques.

Una cadena de bloques puede ser privada o pública, lo cual puede dar acceso a los bloques a nodos específicos de la red, lo cual puede asegurar la privacidad de la cadena en caso de poseer datos sensibles a todos los usuarios, se utilizan firmas digitales para firmar cada bloque, valiéndose de llaves públicas y privadas para firmar los bloques y que puedan ser verificados sólo por los nodos seleccionados.
La Figura **3-1** muestra la estructura de una cadena de bloques.

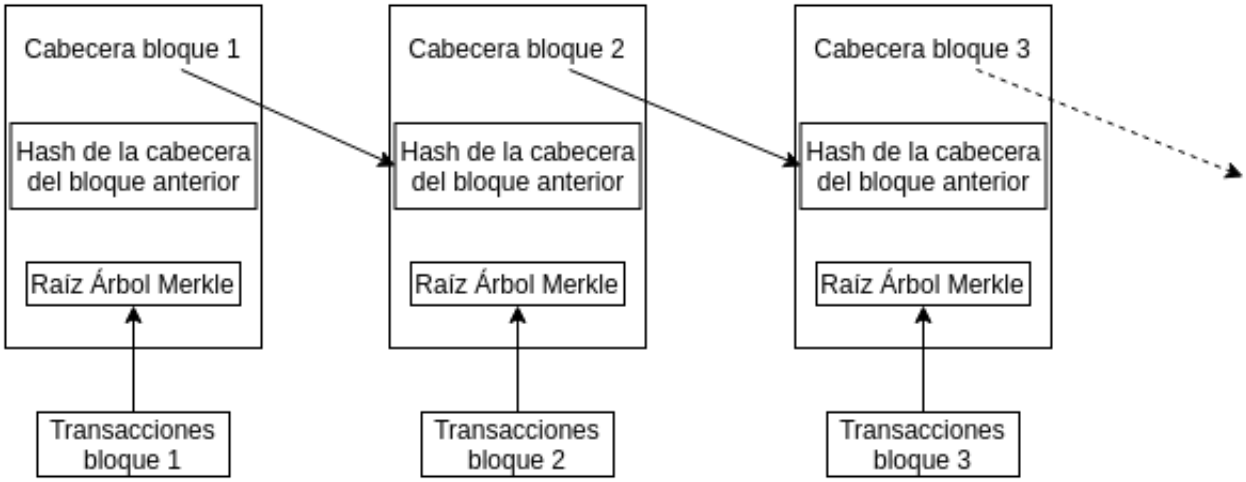

Figura 3-1: Representación gráfica de una cadena de bloques.
Recuperado y traducido de: *https* : *//bitcoin.org/en/developer − guide*

3.2. Propiedades deseables

Para construir un libro mayor digital para uso en un ambiente como la red internet y donde los participantes no tienen relaciones de confianza, la estructura de datos (cadena de bloques), debe contar con algunas propiedades deseables[40]:

1. El libro mayor debe ser inmutable, y de manera más precisa, la estructura de datos únicamente debe permitir agregar datos (append-only), no es posible modificar, remover o reordenar datos existentes.

2. Debe haber una manera de obtener un resumen criptográfico del estado de la estructura en cualquier momento.

3. Un resumen criptográfico es un dato de tipo string, corto, que evita tener que almacenar la estructura de datos completa. Si la estructura de datos fue manipulada de alguna manera, el resumen criptográfico resultante cambiaría, por lo tanto se detectaría la manipulación.

La justificación de estas propiedades, es que a diferencia de una estructura de datos regular que está almacenada en una única máquina, el libro mayor digital es una estructura de datos global mantenida colectivamente por un conjunto de participantes sin relaciones mutuas de confianza.

Este funcionamiento contrasta con otro enfoque para decentralizar libros mayores digitales, en donde muchos participantes mantienen libros mayores locales y es responsabilidad del

usuario hacer consultas en el conjunto de libros mayores para resolver cualquier conflicto.

3.3. Antecedentes académicos

La estructura de la cadena de bloques de la criptomoneda Bitcoin, es tomada prestada con unas mínimas modificaciones, de una serie de artículos de Stuart Haber y Scott Stornetta, escritos entre 1990 y 1997[44].

El trabajo de Haber y Stornetta está relacionado con los problemas del marcado de tiempo (timestamp) de documentos, orientados a construir un servicio de tipo "notaria digital". Para patentes, contratos de negocios y otros documentos de valor, es deseable registrar el tiempo de creación de un documento en un momento dado y no más tarde.

La noción de documento en el trabajo de Haber y Stornetta es bastante general y podría ser de cualquier tipo de dato; aunque mencionan transacciones financieras como una potencial aplicación, no era el foco de la publicación.

En una versión simplificada de la propuesta de Haber y Stornetta, los documentos están constantemente siendo creados y transmitidos. El creador de cada documento establece un tiempo de creación, firma el documento, coloca la marca de tiempo y un apuntador al documento previamente transmitido.

Este documento previo, a su vez, apunta a su predecesor, de manera que los documentos forman una cadena con apuntadores en sentido inverso al tiempo. Un usuario externo no puede alterar el mensaje con su marcado de tiempo ya que está firmado por el creador, y el creador no puede alterar el mensaje sin también alterar la cadena entera de mensajes que le suceden.

Si se obtiene un único item de la cadena a través de una fuente confiable (otro usuario o un servicio especializado de marcado de tiempo), la cadena entera hasta ese punto está bloqueada, es inmutable y está temporalmente ordenada.

En artículos sucesivos, Haber y Stornetta introducen otras ideas que hacen que esta estructura de datos sea mas efectiva y eficiente:

1. Los enlaces entre documentos pueden ser creados usando hashes en vez de firmas; el hash es mas simple y fácil de procesar, estos enlaces son denominados apuntadores hash.

2. En vez de indexar documentos individualmente, lo cual puede ser ineficiente si muchos documentos son creados aproximadamente al mismo tiempo. La propuesta es agrupar los documentos en grupos o bloques, donde los documentos en cada bloque tienen esencialmente la misma marca de tiempo.

3. En cada bloque, los documentos pueden ser enlazados entre ellos con apuntadores hash en un árbol binario, denominado árbol Merkle, en vez de una cadena lineal.

De manera casual, Josh Benaloh y Michael de Mare independientemente introducen estas tres ideas en 1993[36], poco después del primer artículo de Haber y Stornetta.

3.4. Bitcoin Blockchain

La criptomoneda Bitcoin usa esencialmente la estructura de datos en los artículos de Haber y Stornetta de los años 1991 y 1997, y hacen reingeniería de sus propiedades de seguridad anadiendo el esquema de comprobación de trabajo (proof-of-work, en inglés).

Bitcoin y su cadena de bloques, puede ser la mejor instanciación conocida de las estructuras de datos de Haber y Stornetta. Sin embargo, no es la primera implementación, al menos dos empresas habían tenido una aproximación. A mediado de la década de 1990, la empresa Surety y en el año 2007, la empresa Guardtime ofrecen servicios de marcado de tiempo de documentos.

3.5. Árboles Merkle

Es una estructura de datos, que recibe el nombre por Ralph Merkle, un pionero de la criptografía asimétrica que propuso la idea en un artículo en el año 1980[39]. El objetivo era producir un resumen para un directorio público de certificados digitales.

La Figura **3-2** ilustra de manera simplificada la estructura de datos propuesta por Haber y Stornetta, que enlaza los bloques marcados de tiempo usando apuntadores hash.

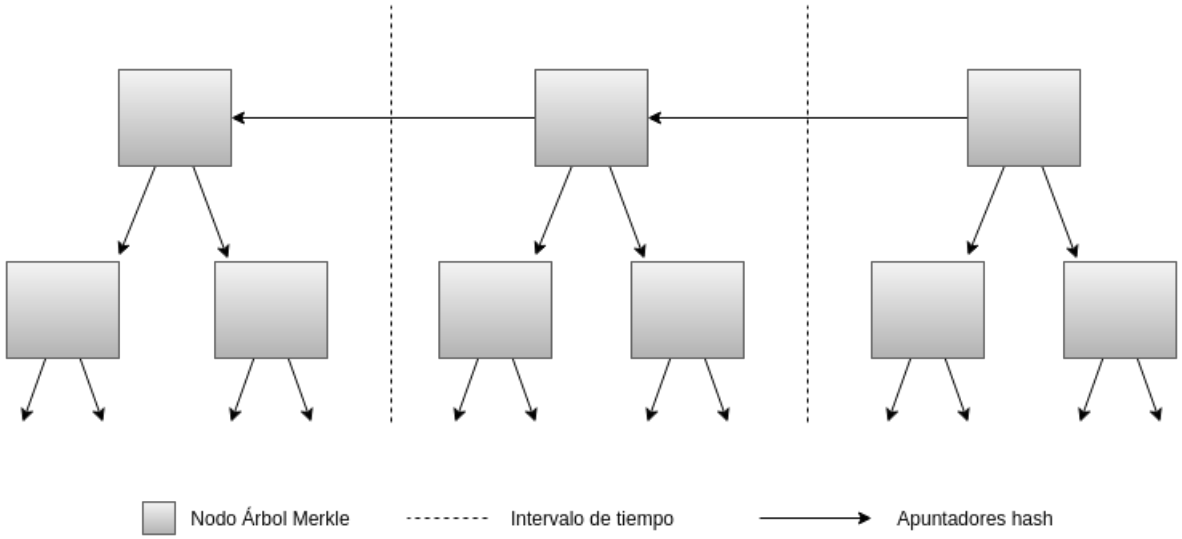

Figura 3-2: Representación simplificada de la estructura de datos que enlaza bloques marcados de tiempo.

El árbol Merkle es un árbol (Generalmente binario, pero no necesariamente) que almacena datos en sus nodos hoja, por cada nodo hoja que contiene datos es generado su código hash y se concatena con el código hash de su nodo hermano, a la concatenación de estos dos hash, se le vuelve a aplicar la misma función criptográfica, la cual se convierte en el nodo padre de dichos nodos hojas, este proceso se repite sucesivamente hasta llegar a la raíz del árbol. Esta estructura es utilizada debido a que el orden de acceso de un árbol es menor que usar una estructura secuencial.

Las transacciones de Bitcoin toman la forma de documentos. En el árbol Merkle de cada bloque, los nodos hoja son transacciones, y cada nodo internamente consiste de dos apuntadores.

Esta estrucutra de datos tiene dos propiedades importantes. Primero, el hash del último bloque es un resumen. Algún cambio a cualquiera de las transacciones (nodos hoja) requerirá propagar los cambios hasta la raíz del bloque, y las raíces de todos los bloques.Por lo tanto, si se conoce el último hash, es posible obtener el resto del libro mayor desde una fuente no confiable y verificar que no ha cambiado.

Un argumento similar establece otra propiedad importante de la estructura de datos, se puede probar eficientemente que una transacción particular está incluida en el libro mayor. El usuario tendría que enviar un pequeño número de nodos en ese bloque de la transacción (este es el objetivo del árbol Merkle), así como una pequeña cantidad de información para cada siguiente bloque. La habilidad para probar inclusión de transacciones eficientemente es altamente deseable por rendimiento y escalabilidad.

La figura **3-3** ilustra en detalle el árbol Merkle en un bloque de la cadena de bloques.

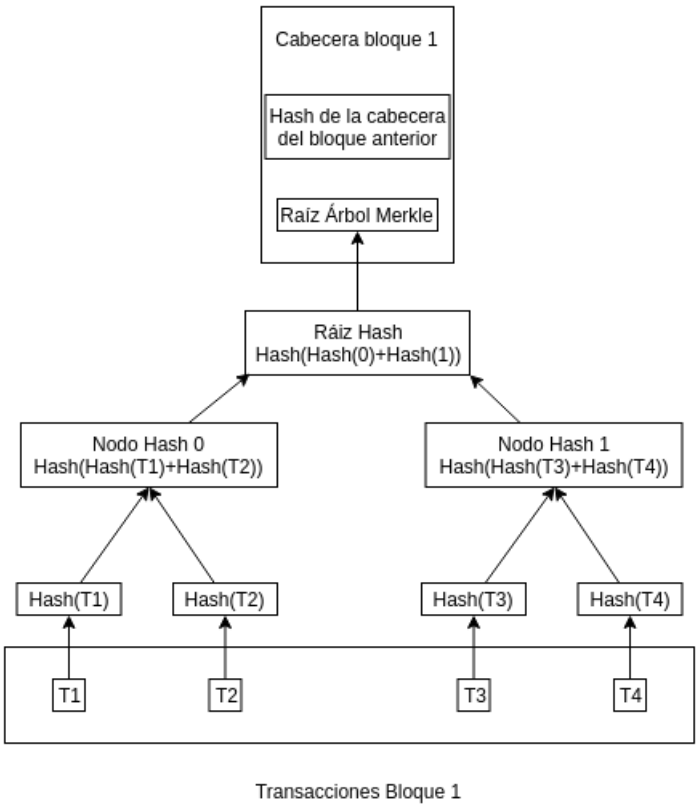

Figura 3-3: Cadena de bloques y el árbol Merkle.
Recuperado y traducido de: *https* : *//bitcoin.org/bitcoin.pdf*

3.6. Prueba de trabajo (Proof-Of-Work)

Es un método de consenso que se popularizó mediante su uso en la plataforma de Bitcoin[40], pero en realidad tiene sus raíces en el sistema Hashcash propuesto por Adam Back para combatir los ataques DDOS y correo spam[31]. Se fundamenta en el hecho de que el emisor de un mensaje debe realizar tareas que tengan cierto tipo de valor (computacional en este caso) para poder enviar su mensaje, de esa manera, un ataque DDOS podría llegar a ser excesivamente caro y si bien no puede neutralizarlo per se, al menos haría que al atacante tuviera que invertir considerables cantidades de poder computacional para saturar algún servidor mediante volumenes masivos de solicitudes, lo mismo ocurre con el correo spam, un emisor de correo spam podría pensárselo dos veces antes enviar miles de correos a distintas direcciones de correo.

Dado el enfoque anterior y orientando esta idea a la cadena de bloques, el proceso involucra resolver un problema computacional, el cual requiere una gran cantidad de poder para resolver, pero un mínimo de poder para validar si la solución encontrada es la correcta. Una vez que se ha invertido el poder computacional para resolver dicho problema, toda la red de la cadena de bloques puede invertir un mínimo esfuerzo para comprobar si el esfuerzo invertido para validar el bloque resolvió el acertijo computacional planteado para validar un bloque.

La resolución del acertijo computacional se relaciona con la obtención de un hash de la solución el cual tenga una estructura definida (Puede ser una cantidad de bits en cero al inicio del hash) por el protocolo, la cual puede variar según la cadena de bloques en la cual se desarrolle esta prueba de trabajo.

3.7. Casos de uso de las cadenas de bloques

Ya establecidas las bondades que pueden facilitar las cadenas de bloques, los escenarios en los cuales puede aplicarse son muy diversos[35]. A continuación se analizan los casos de uso más importantes dónde la cadena de bloques destaca por los beneficios que ofrece:

3.7.0.1. Criptomonedas

Probablemente el caso de uso más famoso de la cadena de bloques. Las criptomonedas, según Usman[33] pueden definirse como bienes digitales que pueden ser intercambiados entre distintos entes. Dichos bienes estan sustentados bajo principios critpográficos que aseguran el flujo transaccional que genera el intercambio de estas monedas digitales, ademas de controlar la creación de bienes adicionales con respecto a las mismas. Existen una gran cantidad de criptomonedas hoy en día las cuales han logrado gran aceptación a nivel mundial y, hasta la

fecha, siguen aumentando su valor.

3.7.0.2. Servicios financieros

Las empresas siempre necesitan hacer compras de grandes cantidades de elementos a crédito para, por ejemplo, conseguir materia prima que se procesa y se vende a otra empresa. Los pagos que se realizan pueden pasar por varios intermediarios antes de que el dinero pueda llegar a su destino final, y por cada intermediario que forma parte de la transacción este gana un pequeño porcentaje. Con la cadena de bloques se forma una línea de tiempo donde puede verse la traza que van dejando los fondos a medida que pasan por los diferentes intermediarios, de manera que una disputa con respecto a una transaccón puede ser fácilmente resuelta observando la cadena de bloques y verificando cada bloque que puede auditarse y resolver cualquier malentendido que pueda presentarse de manera transparente.

Además de la ventaja anteriormente mencionada, si varias empresas deciden implementar su propia cadena de bloques pueden librarse de los intermediarios que manejan sus fondos y así ahorrar grandes cantidades de dinero en impuestos de transacciones que imponen estas organizaciones intermediarias.

Una cadena de bloques semi-privada puede implementarse donde sólo las empresas que formarán parte de la red pueden observar el flujo de los fondos que van desde una organización a otra y auditarlas en caso de algún inconveniente, todo esto ligado también al hecho de que pueden generarse contratos inteligentes entre las empresas que forman parte de la cadena de bloques para que las transacciones de realicen de manera automática y además, de una manera incorruptible y transparente.

3.7.0.3. Servicios de pólizas de seguros

Las empresas aseguradoras necesitan una plataforma que brinde velocidad para solicitudes, verifique pólizas, procese eventos cubiertos por dicha póliza y además, ofrezca una rápida respuesta a los usuarios miembros con el fin de cumplir sus obligaciones a cabalidad. La cadena de bloques ofrece la capacidad de realizar un procesamiento de pólizas automático mediante la implementación de contratos inteligentes en la cadena de bloques, de esta manera, cuando ocurre un evento asegurado, y este proviene de una fuente confiable, la política de seguro es automáticamente activada y realiza el pago según las tarifas especificadas en el contrato inteligente. Este enfoque no sólo brinda una plataforma estable y confiable, sino que también brinda rápida respuesta al usuario lo cual aumenta su satifacción con la empresa.

3.7.0.4. Servicios Gubernamentales

Una considerable parte de las instituciones gubernamentales deben realizar o supervisar grandes cantidades de transacciones las cuales implican grandes volumenes de fondos y podría ser necesario verificar su procedencia y destino.
La cadena de bloques puede ser utilizada para formar una red de entidades gubernamentales confiables las cuales pueden verificar todo tipo de transacciones y además, llevar un seguimiento acertado de todos los eventos que pueden emerger de manera imprevista, muchos ciudadanos pierden sus identificaciones a diario, la identidad de los ciudadanos puede fácilmente representarse en una cadena de bloques y en caso de que un ciudadano pierda su identificación, puede ubicarse rápidamente su identidad siguiendo su traza en la cadena de bloques obteniendo toda la información necesaria para comprobar que dicho ciudadano es quien dice ser.
Cada vez que un ciudadano compra una propiedad y necesita registrarla localmente o regionalmente, puede asociarse este nuevo bien a la cadena de bloques de dicho ciudadano, siempre pudiendo verificar el origen de los fondos con los que lo obtuvo, si realizó un cambio de nombre de alguna de sus empresas o si quiere agregar un socio a un contrato inteligente dentro de la misma cadena.
El mismo principio puede aplicarse para certificados de residencia, partidas de nacimiento, estudios realizados y títulos de propiedad, cada uno de estos valores con su marca de tiempo que le da veracidad a toda esta información.

3.7.0.5. Cadena de Suministros

Las cadenas de suministros en general, son parte de un gran sistema de sistemas, cuando ocurre alguna falla en alguno de estos sistemas a veces puede ser engorroso encontrar el elemento que ocasionó la falla y rectificar se convierte en una tarea complicada. Una cadena de bloques puede brindar detalles de cada etapa de la cadena de suministros junto con marcas de tiempo certeras para ayudar a encontrar fallas en complejas cadenas de suministros, además aporta:

- Confianza incrementada alrededor de toda la cadena de suministros, dado que cada elemento de la red es parte del sistema y no existe un "propietario"de toda la cadena de bloques.
- Optimización del tiempo utilizado para realizar diagnóstico y solucionar fallas en la cadena de suministros.

3.7.0.6. Salud Púbica/Privada

La industria médica necesita sistemas más certeros y eficientes para mantener los registros de sus pacientes, pre-aprobar pagos realizados para intervenciones quirurgícas y realizar compras

de medicamentos vitales para su funcionamiento, la cadena de bloques provee soluciones eficientes para cada uno de estos aspectos.
Los registros médicos electrónicos generalmente se almacenan en sistemas centralizados y su acceso suele ser limitado a la misma red hospitalaria o a agencias de seguros.
La cadena de bloques brinda la historia médica entera de un paciente con una granularidad personilazada según la cadena de bloques implementada a la que podía acceder cualquier hospital que se encuentre dentro de la red, ahorrando así recursos a la hora de que un paciente requiera ser atendido no importa el lugar donde se encuentre, de igual forma puede observarse la evolución de cierto tipo de enfermedades, e incluso actualizar contratos inteligentes para la compra de medicinas imprescindibles para la recuperación de los pacientes de manera inmediata.

3.7.0.7. Internet de las cosas (Internet of things)

Los dispositivos actuales interactúan entre ellos mucho más de lo que sucedía hasta hace unos pocos años, cualquier interacción importante puede ser almacenada en una cadena de bloques privada lo cual facilitaría el monitoreo de estos dispositivos y haría mucho más sencilla la tarea de encontrar fallas en estos sistemas que cada día se vuelven más complejos.

3.8. Contratos inteligentes y Ethereum

3.8.1. Aplicaciones distribuidas (DAPP)

David Johnston y su equipo[28] de desarrollo presentan la siguiente estructura que define una DAPP y sus características:

3.8.1.1. El nacimiento de las DAPPs

Una nueva aproximación está surgiendo para desarrollar aplicaciones altamente escalables y potentes. Bitcoin fue el pionero en esta nueva tendencia, con su características relacionadas a código abierto, conectividad P2P (Red entre iguales o red entre pares, por sus siglas en inglés[43]), criptografía e integración con las cadenas de bloques. Un considerable número de aplicaciones está adaptando este esquema, siendo Ethereum una de las más notables, la cual implementa su propia cadena de bloques para operar en su plataforma.

3.8.1.2. Definición de DAPP

Para que una aplicación pueda considerarse DAPP, debe seguir el siguiente criterio:

- La aplicación debe ser en su totalidad código abierto, debe operar de manera autónoma y sin ningún tipo de control por parte de cualquier organizacón. La aplicación puede

adaptarse a protocolos propuestos en respuesta a las necesidades dinámicas que se presentan a lo largo de su evolución, pero todos los cambios deben realizarse en consenso con sus usuarios.

- Los registros y data de la aplicación deben ser almacenados mediante el uso de criptografía, en una cadena de bloques pública y descentralizada para evitar puntos centrales de fallo.
- La aplicación debe hacer uso de un token criptográfico el cual es necesario para acceder a la aplicación y todas las contribuciones (a mineros) de recompensa deben ser asignadas en dichos tokens.
- La aplicación debe generar tokens acorde a un algoritmo criptográfico estándar que actúa como una prueba de contribución de los nodos a la plataforma de la aplicación (Ya sea Prueba de trabajo u otro algortimo).

3.8.1.3. Bitcoin como DAPP

Bitcoin ha demostrado ser una solución eficiente a los problemas que surgen al momento de implementar una red autónoma puerto a puerto utilizando su propia cadena de bloques. Lo más importante con respecto al tema de interés de la presente investigación, es que Bitcoin se considera una DAPP, por las siguientes razones:

- La plataforma y software de Bitcoin es en su totalidad código abierto, no es controlada por ninguna entidad u organización y todos los datos y registros son públicos a toda el ecosistema.
- Bitcoin genera sus tokens (bitcoins), con un algoritmo predeterminado que no puede ser cambiado, y dichos tokens son necesarios para el funcionamiento de Bitcoin. Los mineros de Bitcoin son recompensados con bitcoins por su trabajo asegurando la integridad de la red.
- Todos los cambios en Bitcoin deben ser aprobados mayoritariamente en un consenso utilizando el mecanismo de Prueba de trabajo.

3.8.1.4. Clasificación de las DAPP's

Existen varias características que definen la clasificación de una DAPP, la clasificación dependerá mayormente de si la aplicación utiliza su propia cadena de bloques o utiliza una cadena de bloques de otra DAPP para lograr sus funciones, en base a lo anterior:

Tipo I : Aplicaciones descentralizadas las cuales implementan su propia cadena de bloques, el ejemplo primordial de este tipo de DAPP es Bitcoin, también todas las criptomonedas nativas (Lite-coin, Binance-coin, etc.) participan en esta clasificación.

Tipo II : Aplicaciones descentralizadas que hacen uso de las cadenas de bloques de las aplicaciones Tipo I para lograr sus funciones. Son protocolos que tienen acceso a tokens para poder desenvolverse en el ecosistema de la aplicación Tipo I.

Tipo III : Aplicaciones descentralizadas que hacen uso de las cadenas de bloques de las aplicaciones Tipo II para lograr sus funciones. Al igual que las Tipo II, aplican protocolos que se desenvuelven en plataformas de aplicaciones de clasificación Tipo II.

Una analogía apropiada para entender las clasificaciones de DAPP's, es la siguiente, la clasificación Tipo I equivale a un sistema operativo tal como Windows, Linux, Mac, etc. Mientras que las Tipo II podrían equivaler a un procesador de texto, o sistema de sincronización de archivos, como Dropbox. Un ejemplo de una DAPP Tipo III sería un software especializado tal como una aplicación que una varios correos electrónicos o una plataforma de blogs que se sincronize con algún editor de texto.
Utilizando la analogía anterior, es altamente probable que existan muy pocas aplicaciones de Tipo I, una cantidad superior de DAPP Tipo II y aún más del Tipo III.

3.8.2. Ethereum

Ethereum es una plataforma para la creación de DAPPs, la misma se presenta como una red puerto-a-puerto(P2P)[43] que tiene integrada su propia cadena de bloques, esta interactúa con la máquina virtual de Ethereum con el fin de proveer un ambiente seguro, integral, transparente y eficiente para construir aplicaciones que dejen registro de sus actividades en dicha cadena de bloques[32].

3.8.3. Ethereum Blockchain

Según Vitalik, principal contribuyente de Ethereum [32], la cadena de bloques de Ethereum es muy similar a la implementada en Bitcoin. La diferencia principal entre Ethereum y Bitcoin con respecto a la arquitectura de su cadena de bloques, es que, a diferencia de Bitcoin, los bloques de Ethereum poseen una copia tanto de la lista de transacciones como del estado más reciente de la cadena. Adicional a lo descrito anteriormente, dos valores que corresponden al número de bloque y la dificultad, respectivamente, los cuales son almacenados también en el bloque. El algoritmo de validación básica de bloques obedece el siguiente flujo:

- Validar si el apuntador al bloque anterior existe y además, es válido.
- Verificar si la marca de tiempo es mayor a la marca de tiempo del bloque referenciado anterior y menor a 15 minutos en el futuro.
- Verificar si el número del bloque, dificultad, raíz de la transacción, raíz del bloque huérfano más cercano y límite de GAS(Ver Sección 3.8.7), son válidos

- Verificar si la Prueba de trabajo del bloque es válida
- Sea $S[0]$ el estado final del bloque anterior.
- Sea Tx la lista de transacciones del bloque, con n transacciones:

  $\forall i \in \{0, 1, 2,...,n\} \rightarrow S[i+1] = \text{APLICAR}(S[i], Tx[i])$

  Si alguna aplicación resulta en error, o si el GAS total consumido hasta el momento en el bloque excede el límite de GAS establecido, retornar un error.
- Sea S_FINAL $= S[n]$, ahora agregando la recompensa del bloque pagada al minero.
- Validar si la raíz del Árbol Merkle del estado S_FINAL es igual al estado final encontrado en la cabecera del bloque. Si esto ocurre, el bloque es válido, de lo contrario, no es válido.

En la Figura **3-4** se observa una representación de la cadena de bloques de Ethereum.

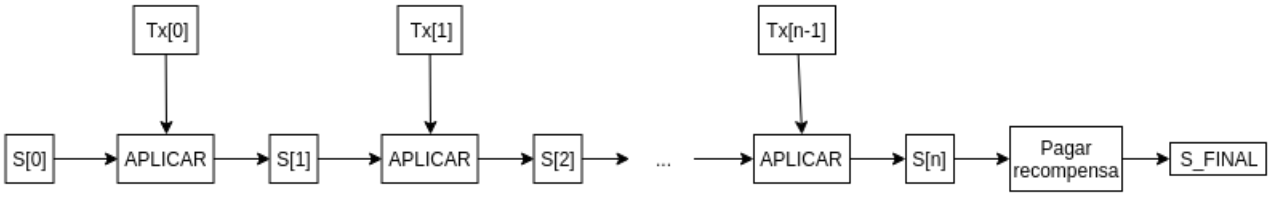

Figura 3-4: Representación gráfica de la cadena de bloques de Ethereum.

Con respecto a Ethereum y su cadena de bloques, para los efectos dentro de esta investigación, se pueden ubicar estos elementos como las bases que darán forma al resto de la herramienta, se puede apreciar la representación de estos elementos en la parte baja de la arquitectura (Figura **3-5**).

3.8.4. Máquina Virtual de Ethereum

En el corazón de Ethereum reside la Máquina Virtual de Ethereum, que puede ejecutar algoritmos arbitrariamente complejos. En Ciencias de la Computación se conoce como un sistema Turing completo, un sistema Turing completo es aquel que tiene un poder computacional equivalente a la máquina de Turing universal[8].

Los desarrolladores pueden crear aplicaciones distribuidas que se ejecutan en la Máquina Virtual de Ethereum utilizando lenguajes de alto nivel como JavaScript y Python.

Cada nodo perteneciente a la red de Ethereum utiliza la Máquina Virtual de Ethereum para realizar las interacciones con la cadena de bloques.

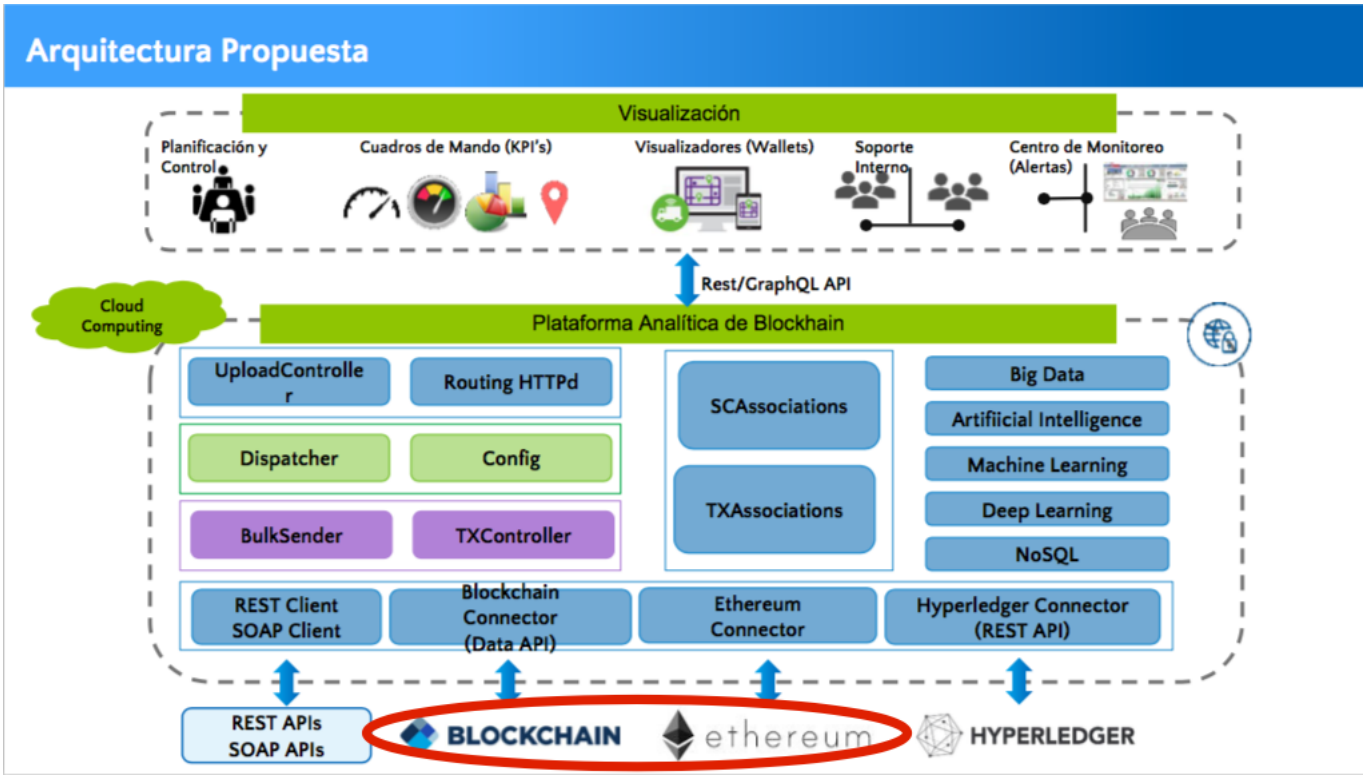

Figura 3-5: Ethereum y su cadena de bloques en la arquitectura planteada.

3.8.5. Tipos de cuentas en Ethereum

Las interacciones entre usuarios dentro de la red Ethereum se realizan a través de dos tipos de cuentas soportadas por la plataforma, las primeras son las cuentas externas (External Owned Accounts, en inglés) y las cuentas de contrato (Contract accounts)[1].
Las cuentas externas, son cuentas controladas externamente, las mismas presentan las siguientes características:

1. Tienen balance en ether.
2. Enviar transacciones (Transmitir ether o activar el código de un contrato inteligente).
3. Es controlada por llaves privadas.
4. No tienen código asociado.

Las cuentas contrato cuentan con estas características:

1. Tienen balance en ether.
2. Posee un código asociado.
3. La ejecución del código es disparada por transacciones o llamadas desde otros contratos

4. Pueden llamar a otros contratos.

Todas las acciones en la cadena de bloques de Ethereum es puesta en marcha por transacciones realizadas por cuentas externas. Cada vez que un contrato recibe una transacción, el código es ejecutado e instruido por la entrada recibida en la transacción, el código es ejecutado por la máquina virtual de Ethereum en cada nodo participante en la transacción como parte de la verificación de nuevos bloques.

3.8.6. Transacciones

La documentación oficial de Ethereum[32] describe una transacción como un paquete de datos seguro que almacena un mensaje a ser enviado por una cuenta externa de Ethereum. Las transacciones contienen

1. Nonce del emisor de la transacción.
2. Dirección de destino.
3. Firma que identifica al emisor del mensaje.
4. El monto de ether que se transfiere desde el emisor del mensaje a la dirección destino.
5. Un campo opcional para datos.
6. Un valor denominado $STARTGAS$ o $GASLIMIT$ (Definido por el emisor), que representa la capacidad máxima de cómputo (medido en instrucciones) que la ejecución de la transacción puede realizar.
7. Un valor denominado $GASPRICE$ (Definido por el emisor), que representa la cuota (en wei) que paga el emisor por instrucción ejecutada

Los primeros cuatro elementos de la transacción estan implícitos en cualquier criptomoneda el campo opcional de datos no tiene una función esperada por defecto, sin embargo, gracias a la máquina virtual de Ethereum el campo de datos de una transacción puede utilizarse como entrada en la ejecución de un contrato inteligente. En la Figura **3-6** se observa una representación de una transacción en Ethereum.

Los campos $STARTGAS$ y $GASPRICE$ son fundamentales para el modelo anti-$DDOS$ de Ethereum. Para poder blindarse contra accidentes o ataques hostiles contra la red, cada transacción debe especificar una cantidad finita de cómputo que será realizado por la máquina virtual. La unidad mínima de cómputo utilizada por un contrato es conocida como GAS.

Transacción en Ethereum

Nonce	Número que representa la cantidad de veces que la presente cuenta ha emitido una transacción.
To	Dirección a la que va dirigido el ether enviado
Value	Cantidad de ether a enviar
GasPrice	Cantidad de ether dispuesta a pagar por cada unidad de gas consumida en la transaccioón
StartGas/GasLimit	Unidades de Gas que la transacción puede consumir
v	Piezas criptográficas que son utilizadas para generar la dirección del emisor de la transacción. Son generadas con la llave privada del emisor.
r	
s	

Figura 3-6: Representación gráfica de una transacción en Ethereum.

3.8.7. Gas - Wei

El GAS puede representarse como el 'combustile' de una transacción en Ethereum, usualmente, una instrucción computacional cuesta 1 GAS, pero algunas operaciones tienen un costo más elevado de GAS que otras, debido a su complejidad y necesidad de consumir más poder de cómputo para ser realizadas o por la necesidad de almacenar mayores cantidades de datos en la cadena de bloques. Existe una tasa adicional de 5 GAS por byte en los de datos de la transacción. La intención de este sistema de tasas, es que un atacante tendrá que invertir una cantidad de GAS proporcional a los recursos que utilizará en la red, incluyendo, cómputo, almacenamiento e iteraciones.
Wei representa una notación de Ether (1 wei = 0.000000000000000001 $ether$), las tasas mencionadas son pagadas por el emisor de la transacción.

La Figura **3-7** representa un ejemplo del uso de *GAS*.

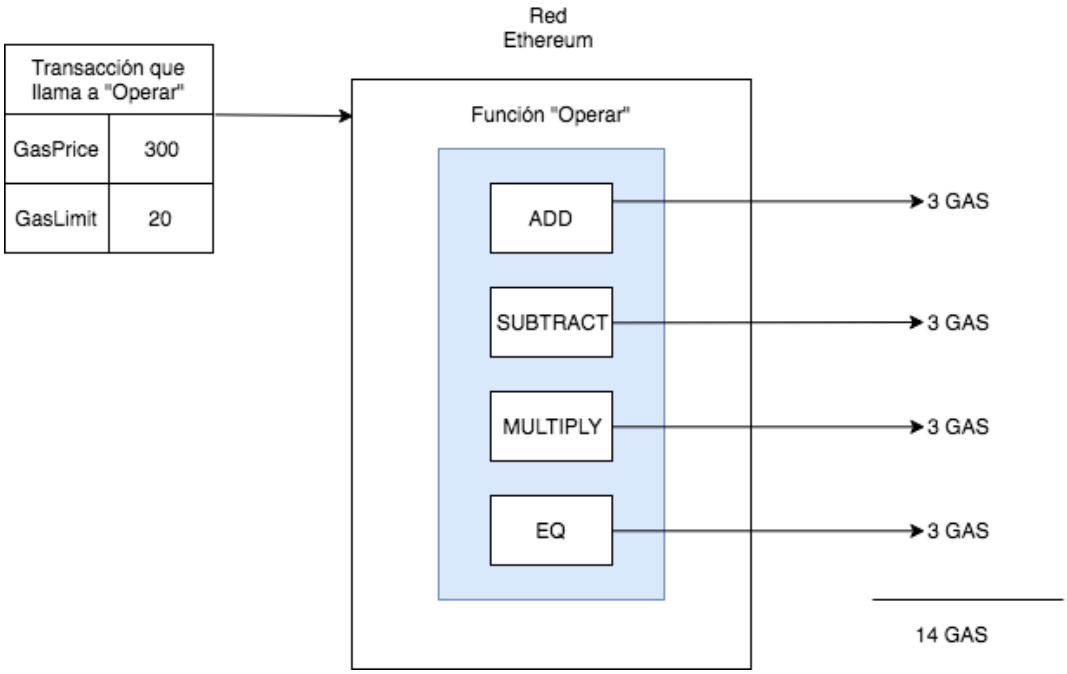

Figura 3-7: Ejemplo de costos de *GAS*.

La Figura **3-8** representa el costo total de una transacción.

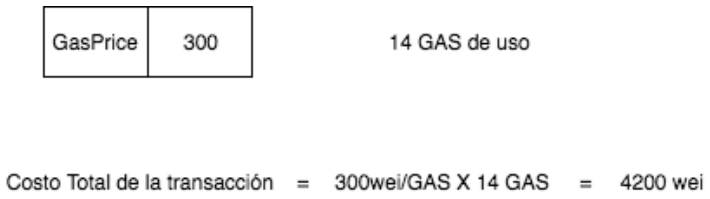

Figura 3-8: Ejemplo de costo final de una transacción

3.8.8. Minado

Una vez una transacción ha sido validada por la red, pasa a un estado "pendiente", cuando una transacción está en este estado, reside en una piscina de transacciones las cuales son seleccionadas por los nodos conocidos como mineros (miners, en inglés), estos nodos, se encargan de ejecutar estas transacciones con la finalidad de ganar las tasas establecidas por el cómputo proporcionado.

Cuando una transacción es seleccionada para ser minada, se le asigna un *nonce* (No debe confundirse con el *nonce* de la transacción), este *nonce*, es el valor a ser iterado para generar un hash válido que cumpla con el requisito de la red.

3.8.9. Tiempo de minado y dificultad del bloque

El tiempo de minado establecido por la red para una transacción es de 15 segundos aproximadamente, este tiempo depende de la dificultad actual del bloque, la dificultad del bloque es la cota del hash generado por el *nonce* del bloque, en caso de que el tiempo sea mayor a 15 segundos, la dificultad del bloque baja, en caso de que sea menor a 15 segundos, la dificultad aumenta. La Figura **3-9** muestra un ejemplo de minado de una transacción con dificultad 100.

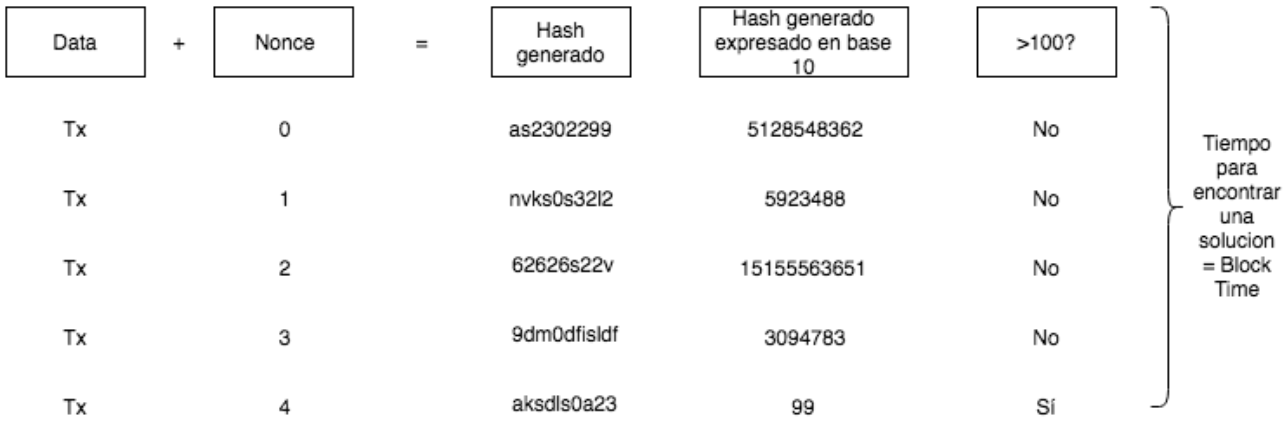

Figura 3-9: Ejemplo minado de una transacción con dificultad 100

Una vez que se encontró el *nonce* que valida la transacción, el nodo minero realiza un anuncio a toda la red anunciando que la transacción fue 'minada' y se procede a almacenarla en la cadena de bloques de Ethereum.

3.8.10. Ether

Según la organización oficial de Ethereum [30], el Ether se define como un elemento necesario para poder realizar operaciones de aplicaciones distribuidas en la plataforma Ethereum. Es la forma de pago realizada por los clientes de la plataforma a las máquinas que ejecutan las operaciones solicitadas.
La primera emisión de Ether fue realizada en el año 2014, arrojando los siguientes resultados:

- 60 millones de Ether emitidos a los contribuyentes de la primera preventa.
- 12 millones de Ether del punto anterior, fueron destinados al fondo de desarrollo, la mayoría siendo asignado a los primeros desarrolladores de la plataforma y el resto a la fundación Ethereum.
- Consenso en el que se generan 5 Ether por bloque, asignados al que minó dicho bloque.

- Entre 2 y 3 Ether son asignados a los mineros que también minaron dicho bloque, pero que su solución no fue incluida en la cadena de bloques (Conocidos como bloques huérfanos o $aunt/uncle - blocks$).

El punto anterior no implica que el Ether sea un recurso infinito, según el acuerdo entre todos los entes participantes en el desarrollo de la plataforma, la emisión de Ether está acotada por 18 millones de Ether al año. Esto significa que, mientras la emisión máxima está acotada, la inflación relativa se disminuye cada año. En teoría, si esta emisión se mantiene de forma indefinida, en algún punto el rango de generación de nuevos *tokens* creados cada año alcanzaría el monto promedio de *tokens* perdidos anualmente (Por falta de uso, pérdida accidental, muerte del propietario, etc.) y finalmente, se alcanzría un equilibrio.

El Ether está destinado a ser utilizado por todos los usuarios que quieran construir aplicaciones que interactuen con la cadena de bloques de Ethereum y además, por todos los usuarios que quieran acceder e interactuar con contratos inteligentes en la cadena de bloques de Ethereum.

3.8.11. Contratos Inteligentes(Smart Contracts)

De todos los datos que pueden guardarse en una cadena de bloques, los contratos inteligentes vienen siendo una de las adiciones más atractivas para lo mayoría de los casos de uso. Un contrato inteligente, haciéndo una analogía con su contraparte física, es un acuerdo automatizado y ejecutable[34].
Son una serie de promesas que son especificadas de manera digital, que incluye los protocolos mediante los cuales las partes involucradas deben realizar dichas promesas. Mientras que es automatizado por algún elemento computacional, puede o no requerir la intervención de control humano. Dicho contrato representa una relación entre dos o más entes que se realiza bajo ciertas condiciones, la ventaja en el uso de contratos inteligentes, radica en que estan enteramente supervisados tanto por entidades legales que los construyen, así como cuerpos computacionales que los ejecutan, haciéndolos a prueba de fraudes.

En lo que a Ethereum se refiere, un contrato inteligente, habita en el ecosistema de su cadena de bloques, el mismo posee una serie de instrucciones que deben ejecutarse a medida que se cumplen las condiciones establecidas en él, cada una de estas instrucciones tiene un costo en Ether, es decir, un contrato elaborado bajo una mala praxis en términos de programación podría ser excesivamente costoso en términos de Ether, por ejemplo, aplicar una función de adición de dos números enteros, es mucho menos costosa en Ether que aplicar una función criptográfica que calcule el hash de un serial de identificación. Es de notoria importancia señalar que las transacciones poseen una cantidad máxima de Ether ($GASLIMIT$) que no debe ser superada por la suma de Ether de todas las intrucciones del contrato inteligente, en caso de superar el límite, se hace un rollback de toda la transacción y el emisor de la misma

no recibe el Ether invertido de regreso, en caso de que el Ether invertido en la transacción sea menor al $GASLIMIT$, pero logró cubrir los costos de todas las operaciones, el exceso de Ether es devuelto al emisor de la transacción.

3.8.12. Contratos Inteligentes: Casos de uso

La Cámara Digital de Comercio ya se encuentra trabajando en el uso de contratos inteligentes, la misma, comenta lo siguiente [25]: 'Los contratos inteligentes permiten a individuos poseer y controlar su identidad digital, conteniendo reputación, datos y bienes digitales. Esto permite a los individuos decidir qué datos compartir a sus contrapartes, dando a las empresas la oportunidad de utilizar únicamente los datos necesarios.'
Tomando en cuenta lo mencionado anteriormente, algunos de los casos de uso de los contratos inteligentes más innovadores son:

3.8.12.1. Contratos inteligentes para Seguridad Financiera

Los contratos inteligentes pueden manejar automáticamente el pago de dividendos, divisiones de acciones y gestión de pasivos, mientras que reduce los riesgos operacionales de confiar operaciones a terceros. Entre las bondades que brindan podemos nombrar:

- Digitalizar flujos de trabajo debido a la confianza del almacenamiento en una cadena de bloques.
- Reducir el riesgo de intromisión de terceros en manejo de capital.
- Ciclos de pagos asegurados gracias a las marcas de tiempo.

3.8.12.2. Contratos inteligentes para Procesos Derivados

Los procesos de post-negociación, pueden ser manejados vía contratos inteligentes, eliminando procesos duplicativos de recursos en ambas entidades luego de realizar un acuerdo comercial. Basta con observar los beneficios:

- Asignación de obligaciones de manera automática, mientras que se ejecutan eventos repetitivos una vez terminada una negociaciόm (Por ejemplo, ciclos de pago periodicos).
- Procesamiento de eventos externos o de sucesión (Aprobación de un crédito, por ejemplo).
- Permite la evalución de movimientos financieros en tiempo real, el cual aumenta el nivel de seguridad y confianza, mientras que minimiza errores y disputas.

3.8.12.3. Contratos inteligentes para la Captación de Datos

Organizaciones financieras pueden hacer uso de contratos inteligentes para la recopilación de datos de manera eficaz, eficiente y transparente. Reducción de procesos de auditoría, datos uniformes entre organizaciones y máximos niveles de trasparencia son algunas de las bondades a explotar, algunas otras se enumeran como:

- Integridad de los datos y transparecia, lo cual genera un mercado con más confianza y estabilidad.
- Al ser almacenados en una cadena de bloques, los datos son almacenados en estructuras accesibles para todos los entes que vayan a utilizar estos datos.

3.8.12.4. Contratos inteligentes para Hipotecas

Los contratos inteligentes pueden automatizar los procesos manuales y tediosos que ocurren tras un contrato hipotecario. En este caso el contrato inteligente brinda una conexión entre todas las entidades relacionadas con la hipoteca, lo cual puede generar un proceso a prueba de errores que sea confiable y transparente. Entre otras ventajas se observan:

- Liberación de contratos a compradores una vez que haya pagado la deuda hipotecaria de manera automática.
- Costos y errores minimizados debido a la eliminación de procesos manuales.
- Verificación de pagos y seguimiento de crédito tanto al comprador como al arrendador.

3.8.12.5. Contratos inteligentes en el Ámbito de Seguros

Actualmente, el proceso de asegurar cualquier bien, está sujeto a muchos procesos manuales y tediosos, los cuales pueden agilizarse de una manera óptima mediante el uso de contratos inteligentes. El contrato inteligente, puede guardar los registro de cada póliza, riesgos cubiertos y, combinado con herramientas del tipo Internet de las cosas, puede emitirse una señal de manera autática cuando ocurren algún tipo de accidente, además ofrece:

- Almecenamiento de historia de un asegurado, conteniendo bienes, pólizas, accidentes cubiertos y tipo de bien asegurado.
- Ahorro de recursos debido a la automatización de procesos de verificación.

3.8.12.6. Contratos inteligentes en el Ámbito Médico

El manejo del historial de un paciente, el cual puede ser almacenado en una cadena de bloques y ser leído por cualquier instirución médica con acceso al contrato inteligente para realizar sus pagos de manera automática es un aditivo atractivo a todos los hospitales o organizaciones médicas del planeta, adicionalmente:

- Reducción de costos debido al ahorro de recursos invertidos en instituciones médicas en mantener el historial de un paciente.
- Acceso a datos íntegros de pacientes los cuales al ser almacenados en una cadena de bloques se vuelven confiables, transparentes y seguros.
- Al estar en la cadena de bloques del paciente, su privacidad aumenta de manera considerable.

3.8.12.7. Contratos inteligentes junto a Internet de las cosas

Probablemente uno de los casos más atractivos. La conexión que puede establecerse entre un dispositivo IOT y un contrato inteligente puede disparar alertas, pagos de consumo y monitoreo en tiempo real de dispositivos de una manera segura, confiable y transparente. Entre otras características, también puede evaluarse:

- Integridad en los datos de consumo, asegurando siempre un registro actualizado en caso de que se haya presentado una falla en algún dispositivo IOT conectado al contrato inteligente.
- Previsión de diferentes escenarios que podrían presentarse dependiente del dispositivo, siempre que esté contemplado en contrato inteligente (por ejemplo, qué hacer si un hogar está produciendo exceso de eletricidad).
- Un historial confiable, transparente y seguro de las actividades de diferentes dispositivos pueden quedar almacenadas mediante la vigilacia del contrato inteligente.

4 Capítulo 4 - Tecnologías utilizadas

El presente capítulo describe las tecnologías utilizadas para la construcción de la herramienta desarrollada.

4.1. Lenguajes de Programación

En computación, un lenguaje de programación es cualquier lenguaje artificial que intenta conservar una similitud con el lenguaje humano, el cual, se utiliza para definir adecuadamente una secuencia de instrucciones que puedan ser interpretadas y ejecutadas en una computadora. [14]
Establecen un conjunto de símbolos, reglas sintácticas y semánticas, las cuales rigen la estructura y el significado del programa, junto con sus elementos y expresiones. De esta forma, permiten a los programadores o desarrolladores, poder especificar de forma precisa los datos sobre los que se va a actuar, su almacenamiento, transmisión y demás acciones a realizar bajo las distintas circunstancias consideradas. Usualmente se clasifican en interpretados y compila- dos, en el cual los compilados tienen un compilador específico que obtiene como entrada un programa y traduce las instrucciones las cuales pueden servir de entrada para otro intérprete o compilado y los interpretados tienen un intérprete específico que obtiene como entrada un programa y ejecuta las acciones escritas a medida que las va procesando.

4.1.1. JavaScript

Según la fundación Mozilla[13], JavaScript es un lenguaje ligero, interpretado, orientado a objetos con funciones de primera clase, más conocido como el lenguaje de script de páginas web, pero también, utilizado en muchos entornos tales como NodeJS.

4.1.2. Solidity

Solidity es un lenguaje de programación de alto nivel orientado a contratos, su creación fue influenciada por C++, Python y JavsScript, está diseñado para funcionar sobre la Máquina Virtual de Ethereum[27].
Solidity es fuertemente tipado, hace uso de herencia, bibliotecas y tipos complejos creados por el usuario.

4.2. Remix

Remix es un ambiente de desarrollo para crear código escrito en Solidity, el mismo se vale del navegador para brindar una interfaz al usuario programador para que puedra realizar prototipos de contratos sin la necesidad de hacer un despliegue en alguna red local o de producción[22].

4.3. Infura

Infura[12] es un servicio el cual simplifica la tarea de desplegar contratos en redes de Ethereum sin la necesidad de configurar un nodo propio, tarea que es bastante compleja de realizar y no es el punto central de la presente investigación.

4.4. MetaMask

MetaMask es un $plug - in$ de código abierto[15] que permite al navegador conectarse, en el caso de esta investigación a la red Ethereum sin tener que pasar por todo el proceso de creación de un nodo Ethereum.

Está principalmente diseñado para usuarios finales, debido a que permitirá la confirmación o rechazo de la salida de transacciones desde un cliente web a la red.

La misma provee una plataforma de autenticación basada en $Mneumonics$ para el manejo de direcciones de Ethereum y adicionalmente, provee la posibilidad de conectarse a las redes de prueba de Ethereum (Rinkedby, Ropsten y Kovan) para realizar pruebas en ambientes de producción sin tener que invertir fondos en adquirir ether para realizar pruebas.

4.5. NodeJS

Es un entorno de ejecución para JavaScript, está diseñado para construir aplicaciones de red escalables. Fue creado con el enfoque de ser útil en la creación de programas de red altamente escalables, como por ejemplo, servidores web 2.0. Fue creado por Ryan Dahl en 2009.

4.5.1. Web3

Web3 es una biblioteca de código abierto[29] compatible con el ambiente de desarrollo NodeJS la cual permite enlazar direcciones de Ethereum (Mediante MetaMask) con instancias de contratos inteligentes que estén instanciados en una red Ethereum desde el código de un cliente web.

4.5.2. SolC

SolC es una biblioteca de código abierto[26] compatible con el ambiente de desarrollo NodeJS la cual compila códigos escritos en Solidity, y genera los archivos necesarios (ABI y *Bytecode*) para poder acceder a los mismos desde el cliente web e interactuar con las instancias obtenidas mediante las llamadas realizadas con Web3.

4.5.3. Ganache

Ganache es un proveedor de código abierto[9] compatible con el ambiente de desarrollo NodeJS, la cual permite emular una red Ethereum de manera local, esto permite realizar pruebas de funcionamiento sin tener que acceder a una red real de Ethereum.

4.5.4. Mocha

Mocha es un *framework* de NodeJS el cual permite realizar pruebas asíncronas de manera rápida y segura[16].

4.6. React

React es un *framework* del lado del cliente construido para poder utilizar JavaScript en la tarea de elaborar interfaces de usuario[20]. Fue creada por Facebook en el año 2013 y está basado en componentes recicables que manejan su propio estado con variables independientes que interactúan con el DOM[6] de una manera dinámica para poder crear interfaces de una manera rápida y utilizando un lenguaje familiar para la mayoría de los desarrolladores.

4.6.1. NextJS

NextJS es una biblioteca que complementa las funciones de React, ya que brinda una lógica conocida como '*server side rendering*'[19] lo cual significa, que antes de mostrar cualquier elemento en el navegador del usuario, el servidor se encarga de primero renderizar todos los elementos de la interfaz, y una vez que este proceso se completó, se muestra la interfaz completa al cliente mediante el navegador

4.6.2. Semantic UI React

Biblioteca que brinda una serie de elementos visuales estructurados los cuales ayudan a la creación de interfaces atractivas y versátiles para los usuarios[24]. Estas pueden ser configurables mediante JavaScript para brindar experiencias de usuario más cómodas y amigables.

4.7. Justificación de tecnologías

4.7.1. MetaMask

MetaMask se posicionó rápidamente como la opción estándar para la interacción con Ethereum debido a que fue la primera herramienta que permitió dicha interacción de manera segura y sin muchas configuraciones adicionales. Adicionalmente, se mantiene al día con actualizaciones regulares.

4.7.2. NodeJS + Web3

El equipo desarrollador de Web3 buscaba brindar una plataforma que se adaptara a las necesidades actuales del mercado y tuviera una proyección positiva hacia el futuro de Ethereum.

En vista de la gran popularidad de NodeJS en los últimos años, debido a su política de código abierto y además, junto a su integración con el gestos de paquetes NPM, permiten a los usuarios crear sus propias bibliotecas que se valieran de Web3 para hacer la herramienta lo más versátil posible.

Siguiendo esta línea de pensamiento, las bibliotecas Solc, Ganache y Mocha vienen siendo convenciones que permiten la muy fácil integración de elementos vitales que son necesarios para lograr crear un sistema robusto que pueda interactuar con Ethereum.

Dada la arquitectura planteada, el uso de estos elementos vendrían siendo la parte intermedia de la arquitectura, el puente que une la base de la red de Ethereum con la interfaz de usuario final, como se puede apreciar en la Figura **4-1**.

4.7.3. React + NextJS + Semantic UI React

La justificación para el uso de React como $framework$ del lado del cliente, sobre otras opciones como $AngularJS$ y otros, viene dada por el rol que juega Ethereum y su máquina virtual dentro de la presente solución.

De cierta forma, Ethererum vendría a reemplazar a lo que sería un servidor en una aplicación web convencional, en vista de este cambio de paradigma, React, que es exclusivamente un $framework$ del lado del cliente, viene siendo una solución que cubre todas las necesidades que puede presentar esta versión de cliente web.

Adicionalmente, su fácil integración con NodeJS brinda todas las herramientas necesarias para plantear con éxito una solución completa al caso de uso seleccionado.

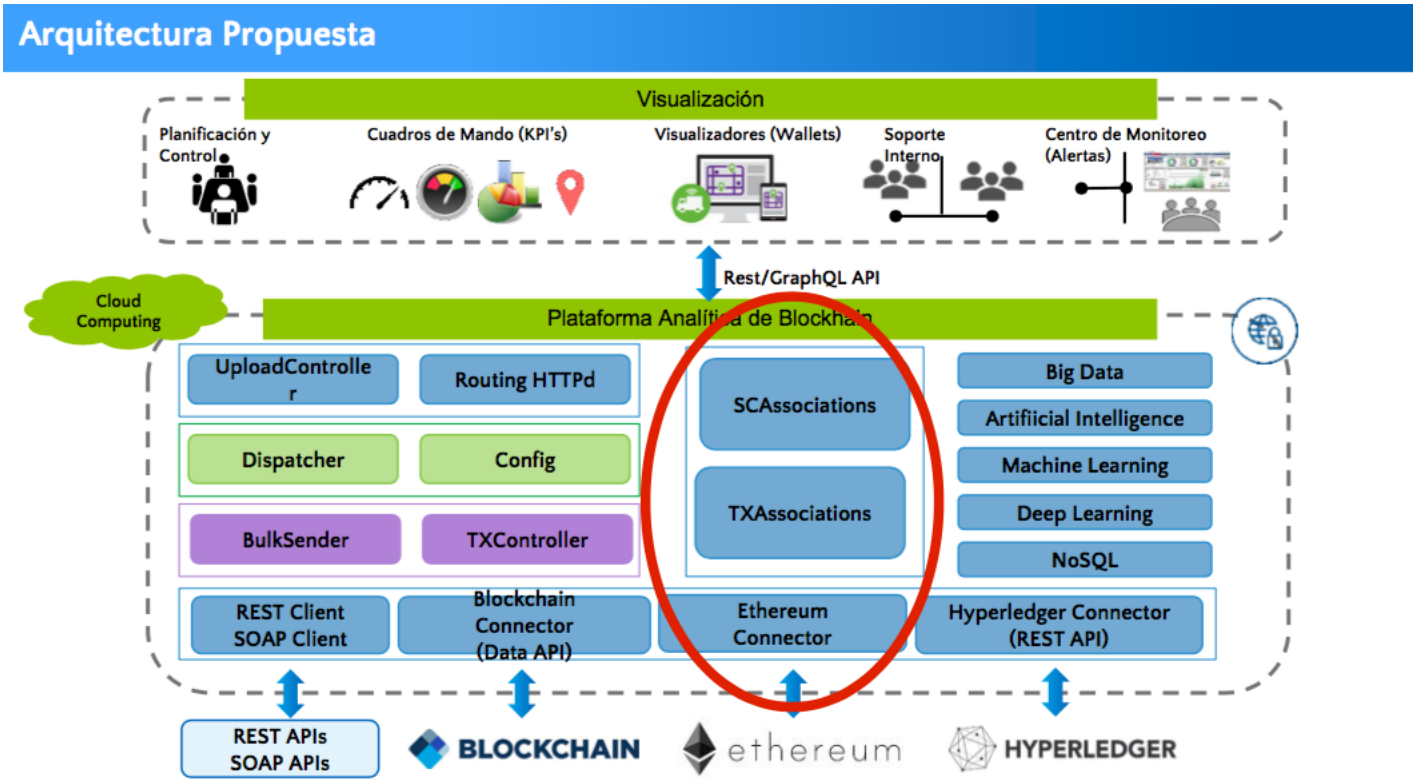

Figura 4-1: Capa intermedia planteada en la arquitectura.

El uso de NextJS nace gracias a la bondad que brinda su '*server side rendering*' la cual se ajusta perfectamente a las necesidades que nacen de la integración con Ethereum. Pedir todos los datos desde la red, crear el DOM, esperar el proceso de minado, realizar consultas asíncronas y luego mostrar el resultado vía el navegador viene siendo el flujo perfecto para este tipo de soluciones.

Finalmente, una vista atractiva al usuario viene siendo tarea fácil gracias a Semantic UI React y todas las bondades que brinda para armar interfaces utilizando React y NextJS.

Todo el trabajo de crear la interfaz de usuario, habilitar el intercambio de información con Ethereum mediante un cliente web y la interacción final del usuario final con la herramienta correponde a la capa superior a la arquitectura planteada (Figura **4-2**).

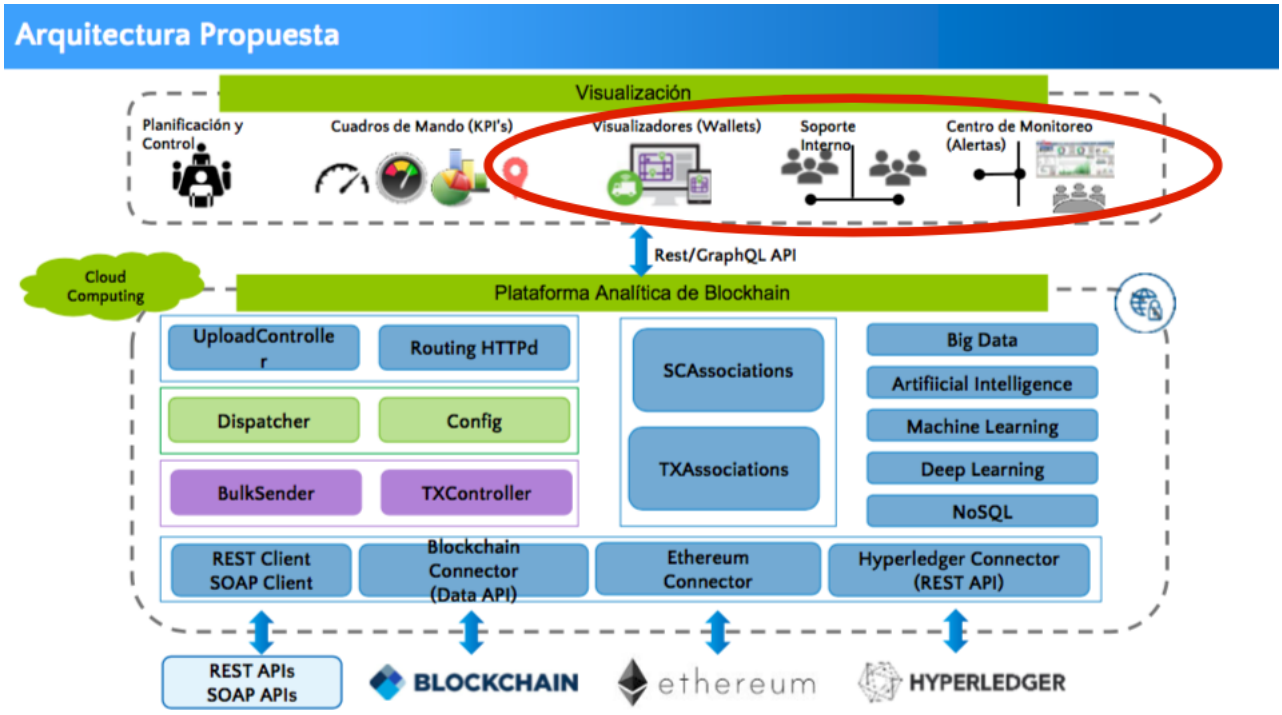

Figura 4-2: Capa final planteada en la arquitectura.

4.8. Almacenamiento

Una de las tareas planteadas para el presente TEG es el de almacenar las interacciones de los usuarios con la cadena de bloques, para ello, es necesario definir las propiedades de algunas bases de datos que puedan ser de utilidad para realizar dicha tarea de forma eficiente.

4.9. Bases de Datos NoSQL

Son un enfoque hacia la gestión de datos y el diseño de base de datos que es útil para grandes conjuntos de datos distribuidos. Los datos almacenados no requieren estructuras fijas como tablas, normalmente no soportan operaciones JOIN, ni garantizan completamente ACID (atomicidad, consistencia, aislamiento y durabilidad).

Son especialmente útiles cuando se necesita acceder y analizar grandes cantidades de datos no estructurados o datos que se almacenan de forma remota en varios servidores virtuales en la nube.[3]

4.9.1. Tipos de Bases de Datos NoSQL

- Orientada a Columnas: Este tipo de bases de datos están pensadas para realizar consultas y agregaciones sobre grandes cantidades de datos. Funcionan de forma parecida a las bases de datos relacionales, pero almacenando columnas de datos en lugar de registros. Algunos ejemplos de base de datos orientada a columnas: Cassandra[4], HBase[10].

- Orientadas a Clave-Valor: Son sencillas de entender. Simplemente guardan tuplas que contienen una clave y su valor. Cuándo se quiere recuperar un dato, simplemente se busca por su clave y se recupera el valor. Algunos ejemplos de base de datos clave/valor: DynamoDB[7], Redis[21].

- Orientadas a Documentos: Son aquellas que gestionan datos semi estructurados. Es decir documentos. Estos datos son almacenados en algún formato estándar como puede ser XML, JSON o BSON. Son las bases de datos NoSQL más versátiles. Se pueden utilizar en gran cantidad de proyectos, incluyendo muchos que tradicionalmente funcionarían sobre bases de datos relacionales. Algunos ejemplos de base de datos orientada a documentos: MongoDB[17], CouchDB[5].

- Orientada a Grafos: Basadas en la teoría de grafos utilizan nodos y aristas para representar los datos almacenados. Son muy útiles para guardar información en modelos con muchas relaciones, como redes y conexiones sociales. Algunos ejemplos de base de datos orientada a grafos: Infinite Graph[11], Neo4j[18].

4.9.2. MongoDB

Es una base de datos ágil que permite a los esquemas cambiar rápidamente a medida que las aplicaciones evolucionan, proporcionando siempre la funcionalidad que los desarrolladores esperan de las bases de datos tradicionales, tales como índices secundarios, un lenguaje completo de búsquedas y consistencia estricta[17].
La misma pertenece a la familia de las bases de datos NoSQL orientadas a documentos, donde se pueden almacenar archivos en formato JSON donde se permite la anidación de los mismos.
Posee un lenguaje de consulta propio del sistema manejador de base de datos, el cual permite realizar cualquier tipo de consulta sin importar el esquema del documento almacenado.
Entre las bondades que brinda MongoDB destacan:

1. MongoDB almacena sus documentos en formatos JSON flexibles, haciéndo que los documentos puedan variar a dependiendo de lo que se quiera almacenar.
2. Trabaja con modelos definidos dentro de la aplicación, lo que permite simpleza al momento de modelos los datos a almacenar.
3. Consultas a la medida (Ad hoc), indexación y agregaciones en tiempo real proveen de herramientas flexibles y potentes para analizar los datos.
4. Es código abierto lo que viene siendo una parte fundamental en la creación de DAPP's.

4.9.3. Justificación de uso MongoDB

MongoDB al ser una base de datos orientada a documentos, que además, se vale del formato JSON para manejar los modelos, viene siendo la convención perfecta para trabajar junto a tecnologías como NodeJS, gracias a su base en JavaScript, el manejo de objetos no tiene variación alguna con respecto a trabajar con los objetos del lenguaje, y luego almacenar dichos objetos en la base de datos.

Adicionalmente, la flexibilidad con la que se almecenan los datos, resulta extremadamente ventajoso, a la hora de cambiar el modelo de los datos que se desean almacenar, si a partir de cierto momento, un tipo de dato no contemplado resulta ser bastante útil, puede realizarse una m ínima modificación en el modelo de datos sin romper la integración del sistema.

4.10. Inteligencia Artificial

4.10.1. Analítica Predictiva

La analítica predictiva es una forma de análisis avanzado que utiliza datos nuevos e históricos para predecir la actividad futura, el comportamiento y las tendencias. Implica la aplicación de

técnicas de análisis estadístico, consultas analíticas y algoritmos automáticos de aprendizaje automático a conjuntos de datos para crear modelos predictivos que sitúen un valor numérico o puntuación en la probabilidad de que ocurra un evento particular.
Las aplicaciones de software de análisis predictivo utilizan variables que pueden medirse y analizarse para predecir el comportamiento probable de individuos, maquinaria u otras entidades[41].

4.10.2. Aprendizaje Automático (Machine Learning)

El aprendizaje automático (Machine Learning, en inglés) es una rama de las ciencias de la Computación que es usada para describir algoritmos tanto de aprendizaje supervisado (Predicción y clasificación) como aprendizaje no supervisado (Clusterizar y detección de comportamiento). Es una rama derivada de la Inteligencia Artificial, que busca desarrollar en las computadoras habilidades de aprendizaje sin ser explícitamente programadas para ello[42].
El aprendizaje automático no es un dominio específico sino un conjunto de dominios que ofrecen diferentes enfoques para resolver problemas complejos. El propósito general del aprendizaje automático es el desarrollo de algoritmos.
Junto a la inteligencia artificial, se enfocan en el desarrollo de algoritmos que pueden enseñarse a sí mismos para adaptarse iterativamente al momento de exponerse a nuevos datos y continuamente mejorar sus resultados.

4.10.3. Aprendizaje profundo (Deep Learning)

El aprendizaje profundo (Deep Learning, en inglés), es una sub-rama del aprendizaje automático que está basado en aprender en diferentes niveles de representación correspondientes a una jerarquía con respecto a características o conceptos, donde los conceptos de alto nivel son definidos en base a los conceptos de bajo nivel y los mismos conceptos de bajo nivel ayudan a construir conceptos de alto nivel.

El aprendizaje profundo es una parte mas amplia de los métodos de aprendizaje automático basado en aprender representaciones. Una observación (por ejemplo, una imagen) puede ser representada de varias maneras (Como un vector de píxeles), pero algunas representaciones pueden hacer mas sencillas algunas tareas de interés (Como el reconocimiento facial en imágenes).

Esta rama del aprendizaje automático se encarga de definir que representaciones son mejores para cierta tarea y como aprender a interpretarla[38].

4.10.4. Minería de Datos (Data Mining)

Surgió como una tecnología que busca ayudar a comprender el contenido de una base de datos, resaltando la importancia de la manipulación de los datos, debido al valor que éstos pueden aportar.
En el momento en que el usuario les atribuye algún significado especial, pasan a convertirse en información. Cuando los especialistas elaboran o encuentran un modelo, haciendo que la interpretación obtenida entre la información y ese modelo represente un valor agregado, es cuando se obtiene el conocimiento. La minería de datos busca patrones, comportamientos, agrupaciones, secuencias, tendencias o asociaciones, que puedan generar algún modelo que permita comprender mejor el dominio para ayudar en una posible toma de decisión. Con respecto a un problema [37].

4.11. Alcance de la Investigación

Definiciones más especificas de los elementos anteriormente descritos se escapan del alcance de la presente investigación debido a que el tema principal es la implementación de una aplicación distribuida que pueda desenvolverse en el ambiente de Ethereum y la tarea de obtener datos de interés de la misma.

La metodología con respecto al procesamiento e interpretación de los datos que puedan ser obtenidos a partir del contenido de las cadenas de bloques queda en responsabilidad de futuras investigaciones con respecto a los datos una vez que ya han sido extraídos de la cadena de bloques.

5 Capítulo 5 - Marco metodológico

5.1. Metodología de desarrollo y justificación

La selección de una metodología de desarrollo viene siendo una tarea fundamental a la hora de crear cualquier solución de la índole de la presente investigación.

En un mundo tan cambiante, dónde las tecnologías son cada vez más diversas y especializadas, las metodologías ágiles brillan por su rápida adaptación a los diferentes paradigmas que nacen a diario en el mundo tecnológico. Sin embargo, el enfoque tradicional, brinda muchas herramientas ya establecidas las cuales gozan de validación a nivel mundial, permitiendo crear soluciones de una manera eficiente.

Con el objetivo de explotar lo mejor de ambos mundos, la adaptabilidad de las metodologías ágiles, y lo certero de metodologías tradicionales, la propuesta para la presente investigación es la metodología Proceso Ágil Unificado[2].

5.2. Proceso Ágil Unificado

Los primeros pasos que concibieron el nacimiento de esta metodología fueron en el año 1999 por Scott W Ambler, como una solución para mejorar la metodolgía RUP. La idea principal, era actualizar RUP[2] para que fuera más ágil debido a todos los cambios tecnológicos que nacían en la época.

En esencia, es una versión simplificada de RUP. Describe de una manera simple los acercamientos para crear software orientado a las soluciones de negocios utilizando herramientas ágiles, pero, basándose en conceptos utilizados en RUP[23].

Se siguen conservando elementos de RUP tales como los modelos, pero ahora no de una manera tan dominante, está más enfocado a práticas ágiles, como por ejemplo: El uso de modelos sigue siendo fundamental para simular las soluciones, pero ahora el desarrollo se divide en iteraciones 'Lo suficientemente buenas', según Ambler.

5.2.1. Fases

1. Inicio (*Inception*): La meta de esta fase, es definir el alcance del proyecto, potenciales arquitecturas para una solución viable y definir los costos asociados a la solución y su vialidad.

2. Elaboración (*Elaboration*): Definir la arquitectura de la solución.

3. Construcción (*Construction*): La meta de esta fase es construir software de una manera simple, pero incremental, que solucione las problemáticas bases planteadas en la fase de Comienzo y luego, mediante iteraciones ágiles, ir puliendo problemáticas secundarias.

4. Transición (*Transition*): Esta fase representa la validación del sistema en un ambiente de producción.

5.2.2. Disciplinas y actividades

Las disciplinas son desarrolladas de una manera iterativa, definiendo al equipo de desarrollo actividades que ayuden a valiar, probar y entregar software que satisfaga la necesidad de la solución planteada, las disciplinas son:

- Modelado: La meta de esta disciplina es entender el negocio de la organización, el dominio del problema e identificar potenciales soluciones.

- Implementación: La meta es transformar el modelado anteriormente concebido en código que pueda ser probado y mejorado mediante iteraciones.

- Pruebas: Definir pruebas que aprueben o rechacen el software implementado, asegurando estándares de calidad. Esto incluye encontrar defectos, y verificar cumplimiento de requerimientos.

- Despliegue: Planear la entrega del sistema y ejecutar un plan para que los usuarios finales puedan acceder a la solución.

- Gestión de configuración: La meta de esta disciplina es controlar el acceso a los artefactos de la solución, sólo el manejo de versiones.

- Gestión de proyectos: Dirigir las actividades que tomarán acción dentro del proyecto, esto incluye la interacción con entes externos para asegurar una entrega del producto a tiempo y dentro del presupuesto acordado.

- Ambiente: La tarea de esta disciplina es asegurar un ambiente adecuado para maximizar las tareas de desarrollo de la solución.

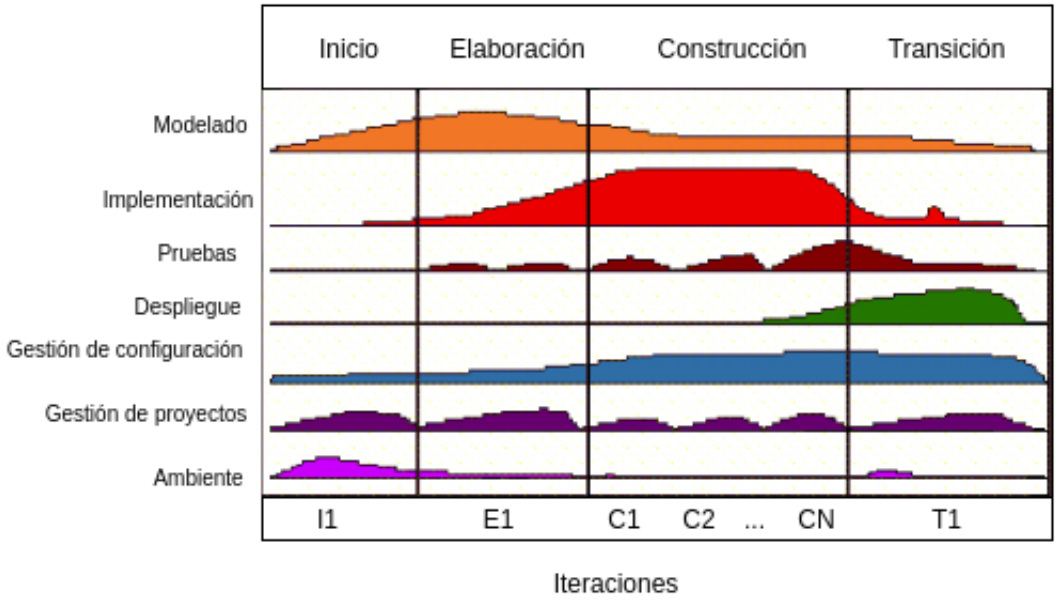

Figura 5-1: Ciclo de vida del Proceso Ágil Unificado.

5.2.3. Iteraciones alrededor del tiempo

En lugar de entregar un sólo proyecto final, se realizan entregas en porciones también conocidas como versiones de prueba, las cuales van evaluando partes del problema que van siendo resueltas, una vez que una versión de prueba es liberada, va a una área de prueba (Conocida también como ambiente de calidad).
La idea de este acercamiento, es ir realizando pre-entregas en intervalos de tiempo cada vez más cortos, con el fin de crear un hábito en el equipo que garantice la agilidad y la robustez del desarrollo a medida que se refinan las pre-entregas, luego de cada cierto número de pre-entregas, se lanza una versión de producción, la cual servirá de base para las siguientes pre-entregas, se puede apreciar un flujo de trabajo en la Figura **5-2** .

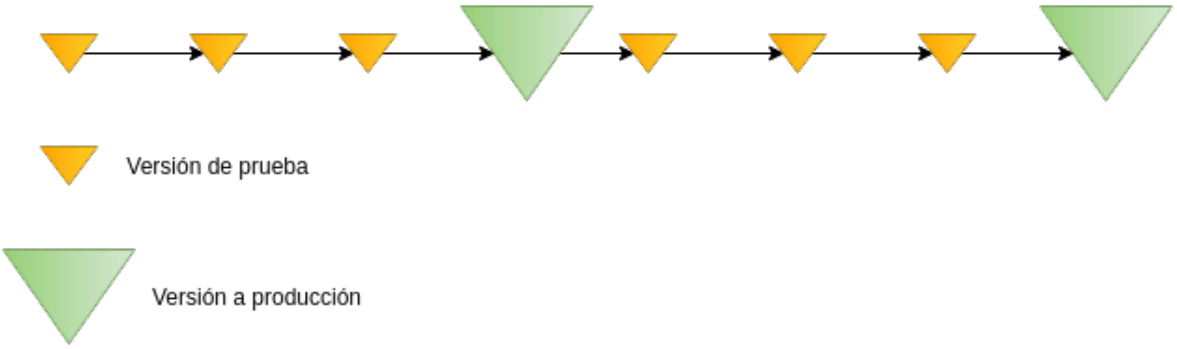

Figura 5-2: Iteraciones de liberación de versiones.

5.2.4. Filosofía del Agile Unified Process

1. El equipo sabe lo que está haciendo: El equipo no requiere una documentación detallada todos los días, realizar guías concretas y cortas es suficiente para asegurar que el equipo vaya en la dirección correcta.

2. Simplicidad: Todos los requerimientos y documentaciones están descritas en páginas cortas al alcance de la mano.

3. Agilidad: Mantener los principios ágiles, pero conservando conceptos básicos de RUP.

4. Mantener el foco en actividades de alto valor: Enforcarse principalmente en actividades que cuentan y son importantes para llegar a la solución.

5. Independencia de herramientas: Dentro de lo necesario para el desarrollo, permitir que el equipo trabaje con las herramientas con las que más se sientan a gusto.

Sin nada más que agregar, el marco aplicativo (Capítulo 6) describirá cómo fue la implementación de esta metodología en la solución a una problemática concreta.

6 Capítulo 6 - Marco aplicativo

6.1. Inicio (*Inception*)

6.1.1. Propuesta de caso de uso para el TEG

6.1.1.1. Solución para evitar malversación de bienes y lavado de dinero en recaudaciones masivas

6.1.1.2. ¿Qué es una recaudación masiva?

Para el presente caso de estudio, una recadación masiva (*crowdfunding* en inglés), se refiere a una campaña colectiva en la cual un gerente propone una idea o modelo de negocios, y luego el público en general puede donar fondos para financiar dicha idea y hacerla realidad.

6.1.1.3. ¿Qué problemas presentan actualmente las recaudaciones masivas?

Algunos de los problemas actuales que presentan las recaudaciones masivas, son la falta de garantías con respecto a los fondos retirados de la campaña, en muchas de ellas se desconoce el destino de los mismos, dejando abiertas sospechas de lavado de dinero. Otro de los problemas que se presentan en este tipo de iniciativas, suele ser que el gerente de una campaña simplemente desaparece con los fondos sin dejar rastro.

6.1.1.4. ¿Cómo pueden Ethereum y los contratos inteligentes atacar estos problemas?

Apoyar este tipo de iniciativas en plataformas como Ethereum le agregan confianza y transparencia, el uso de cadenas de bloques para registrar transacciones, complementado con el pago automático y supervisado mediante contratos inteligentes provee herramientas que reducen eficazmente las problemáticas mencionadas anteriormente.

Adicionalmente, es posible guardar registro de todas las interacciones que se realizan con la cadena de bloques en bases de datos, la finalidad de esta práctica es detectar patrones de comportamiento en este tipo de emprendimientos, mediante el estudio de las campañas, contribuciones, solicitudes, aprobaciones y rechazos.

6.1.1.5. Solución

La propuesta plantea crear una aplicación web que se apoya en el uso de Ethereum y contratos inteligentes para hacer más transparentes y confiables las donaciones y uso de dichas donaciones en campañas de requerimientos de fondos.

En un primer acercamiento, un gerente, se encarga de crear una instancia del contrato inteligente mediante el cliente web.

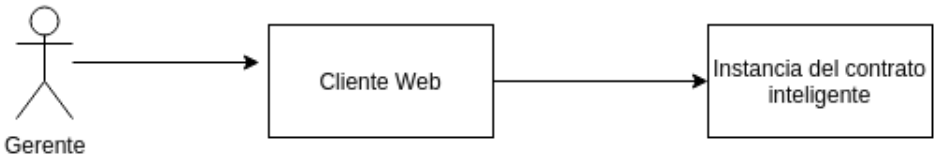

Figura 6-1: Gerente crea una instancia de contrato inteligente mediante un cliente web.

En el cliente web, en la ventana de creación de campañas, el gerente puede parametrizar su campaña y establecer límites o parámetros para su campaña:

- Donación mínima.
- Donación máxima.
- Máximo número de contribuyentes.
- Tasa (%) para aprobar una solicitud.
- Tasa (%) para rechazar una solicitud.

Una vez creada la campaña, los contribuyentes a la misma pueden acceder al cliente web y seleccionar a la campaña a la que desean contribuir y seguidamente, ingresar un monto de ether con el cual quieren contribuir a la campaña, la dirección de los contribuyentes es almacenada en el contrato con el fin de que formen parte ahora de la lista especial de contribuyentes a dicha campaña.

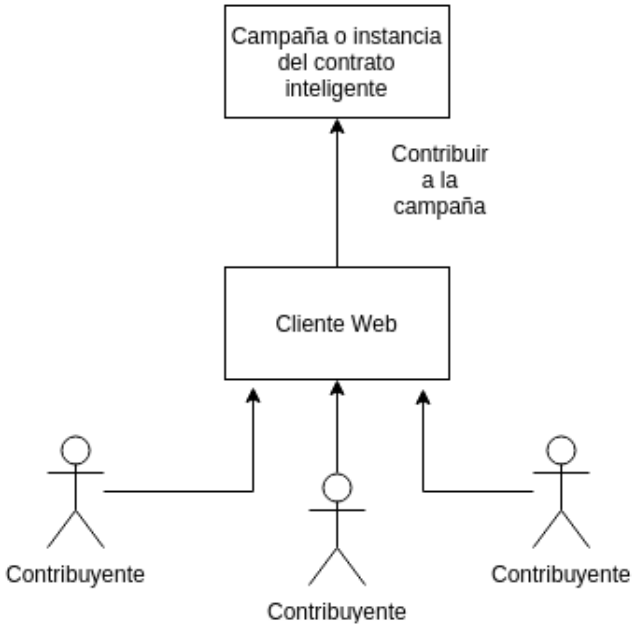

Figura 6-2: Contribuyentes ingresan ether a la campaña mediante el cliente web.

Una vez que una campaña haya recibido fondos de contribuyentes, el gerente se vale del cliente web para crear una solicitud de gasto de la campaña, en la misma debe especificar una descripción de la solicitud, la cantidad de ether que serán enviados, y la dirección destino de los fondos. En la Figura **6-3** se aprecia la morfología de una solicitud:

- 'Comprar baterías' responde a la descripción que debe dar el gerente de la campaña al uso de los fondes que desea retirar.
- 0.5 es el monto especificado que será retirado de la campaña
- '0xnj34...' es la direccón a la cual serán enviados los fondos.

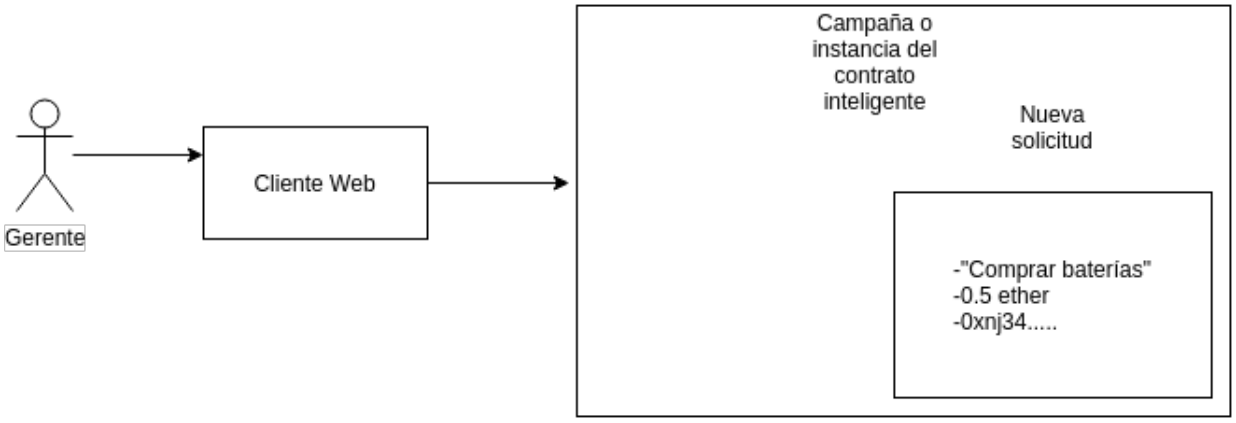

Figura 6-3: Gerente se vale del cliente web para crear una solicitud de gasto.

Una vez creada la solicitud, esta recibe una marca de tiempo, la cual indica la hora exacta en la que fue creada, a partir de ese momento, se calculan 7 días en adelante que representan la vigencia de la solicitud.

Los contribuyentes podrán visualizar mediante el cliente web la solicitud creada por el gerente así como visualizar su descripción, costo, dirección de destino, fecha de creación y expiración. Los contribuyentes pueden aprobar o rechazar que se proceda con ese gasto votando mediante el cliente web. Todo lo anteriormente ocurre siempre y cuando la solicitud se encuentre en vigencia, es decir, que la fecha actual del rechazo o aprobación esté comprendida entre el intervalo de creación y de expiración de una solicitud.

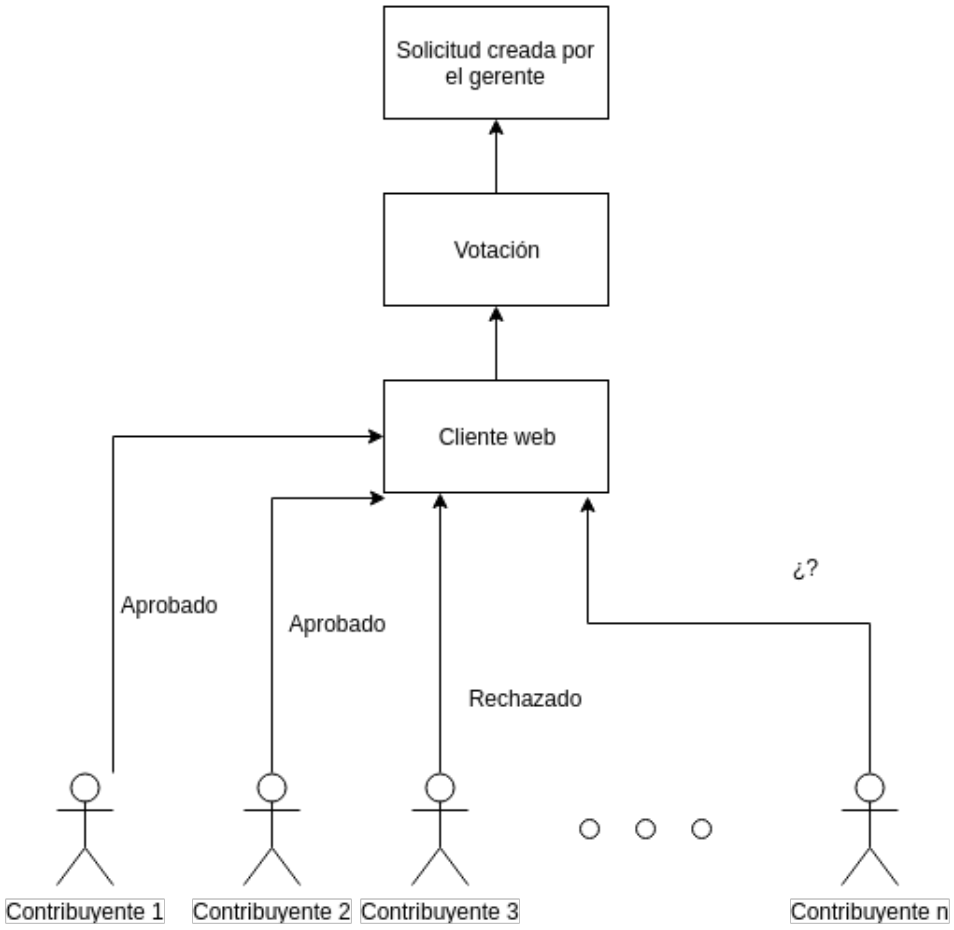

Figura 6-4: Los clientes se valen del cliente web para aprobar o rechazar una solicitud de gasto de una campaña.

Una vez que una solicitud ha obtenido la cantidad de votos de aprobación o rechazo necesarios (Esta tasa es asignada por el gerente al momento de crear la campaña), el gerente de la campaña puede finalizar la solicitud, aprobando o rechazando, dependiendo de cómo haya resultado la votación de la solicitud, si una solicitud cumple con la tasa de aprobación y es finalizada, los fondos son retirados del balance del contrato y son enviados a la dirección destino.

Si la solicitud cumple con la tasa de rechazo y es finalizada, simplemente no se realiza la transacción que va desde el contrato a la dirección destino. El caso de expiración de una solicitud se presenta cuando una solicitud no logró cumplir con la tasa de aprobación o rechazo (El margen de votación es de 7 días), puede ocurrir en casos donde se presenta abstinencia de votos o simplemente las tasas no fueron alcanzadas. Con el fin de no crear puntos muertos en el emprendimiento, si una solicitud expira y es finalizada, se tomará como aprobada y los fondos serán enviados a la dirección destino de la solicitud.

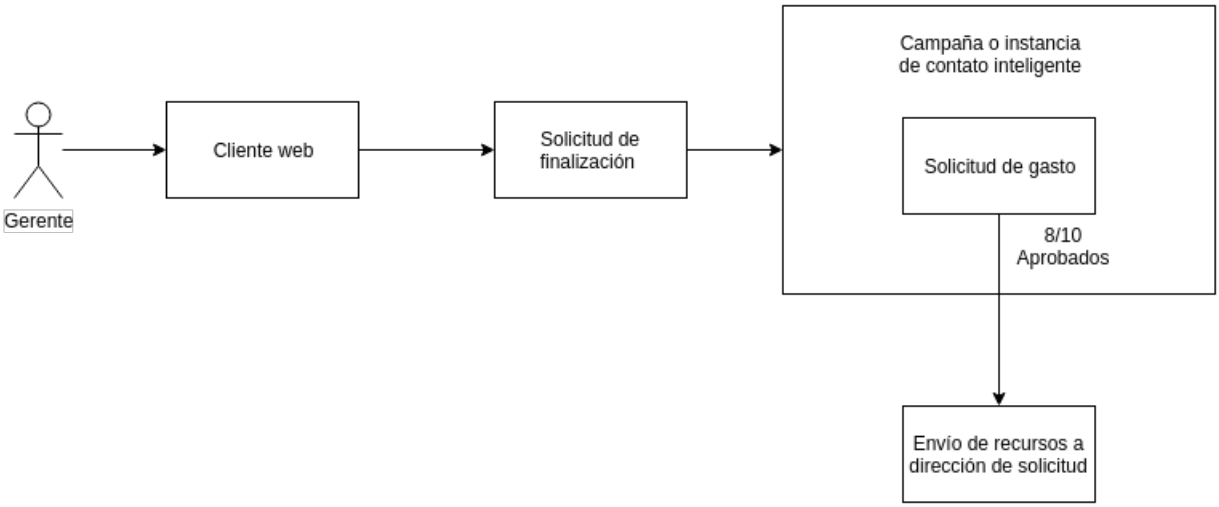

Figura 6-5: Cuando se haya alcanzado la cantidad necesaria de votos, el gerente puede finalizar la solicitud.

Adicionalmente, con el fin de detectar patrones de comportamiento y encontrar datos relevantes en este tipo de emprendimientos, todas las interacciones que se realicen con la cadena de bloques (Crear campaña, contribución, creación de solicitudes, aprobar y rechazar solicitudes) quedan registradas en una instancia de MongoDB con el fin de llevar un registro detallado de las direcciones que interactúan con el contrato inteligente y cómo son dichas interacciones.

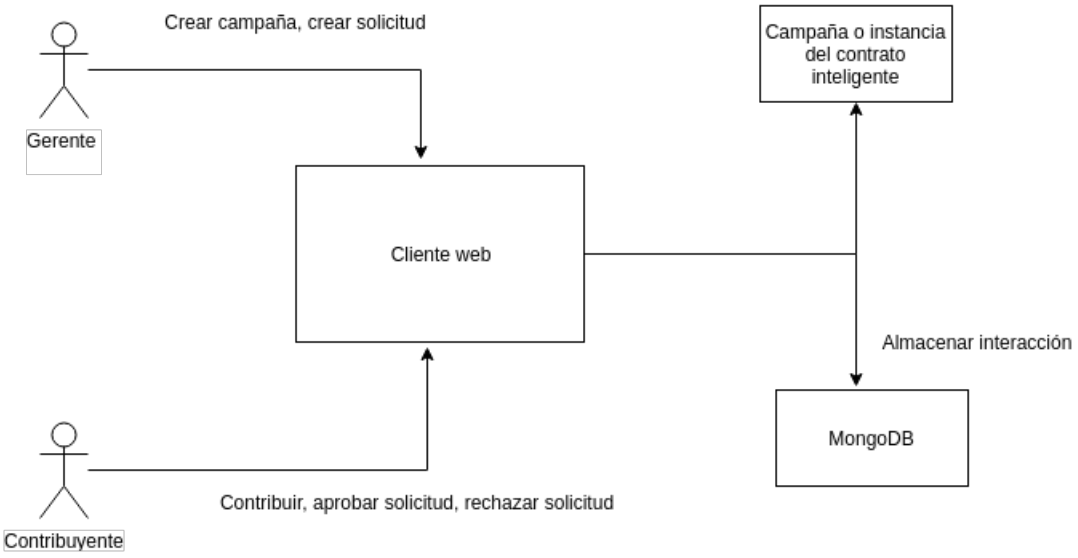

Figura 6-6: Todas las interacciones de parte de contribuyentes y el gerente con la campaña son registradas en una instancia de MongoDB.

Finalmente, se presentan los diagramas de casos de uso tanto como para el gerente como para un contribuyente.

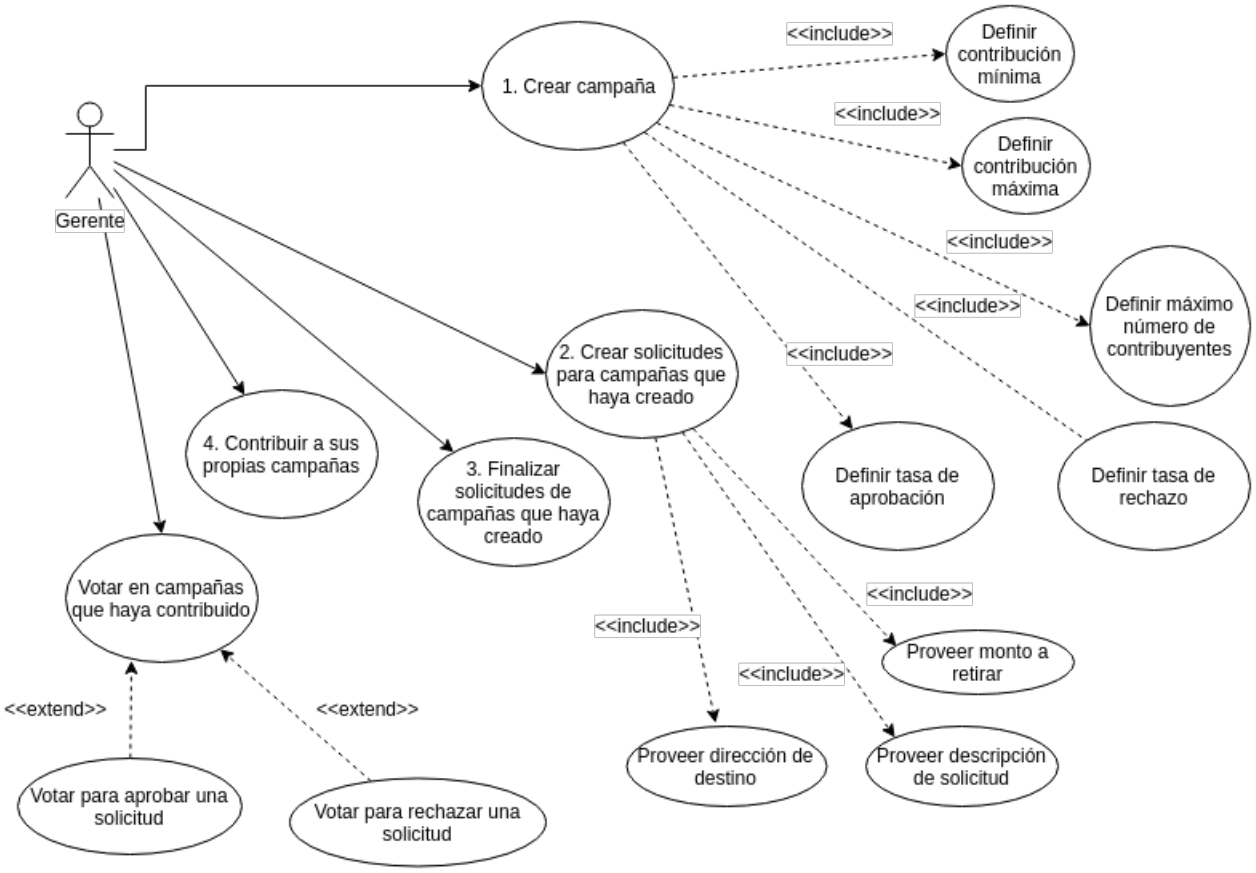

Figura 6-7: Diagrama de caso de uso para el gerente de una campaña.

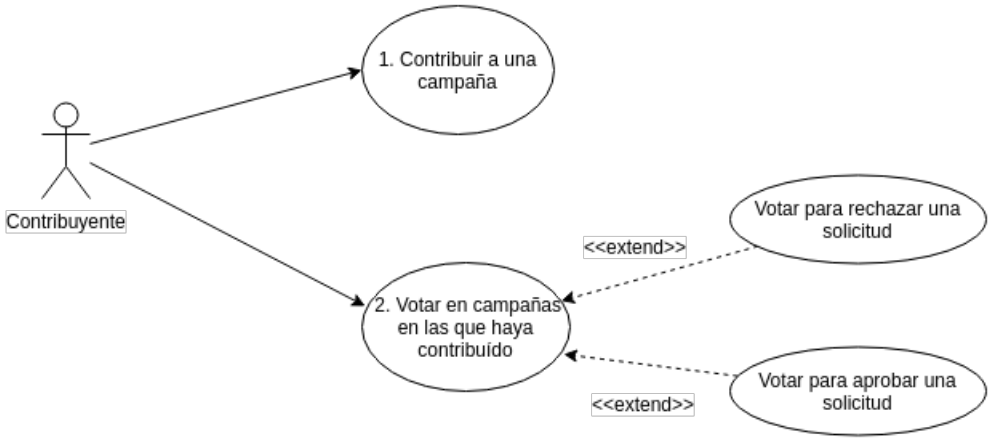

Figura 6-8: Diagrama de caso de uso para contribuyentes de una campaña.

6.2. Elaboración (*Elaboration*)

La arquitectura planteada para solución del caso de uso fue planteada en la introducción del presente trabajo y a sus componentes fueron identificados en los capítulos 3 y 4, respectivamente.

6.3. Construcción (*Construction*)

En esta sección se definirán las iteraciones que darán forma a la solución,

6.3.1. Iteración 0: Acercamiento inicial

En esta primera iteración se realizó el primer prototipo del código fuente, el mismo fue escrito en Solidity, este primer prototipo ataca las problemáticas fundamentales descritas en el caso de uso, pero deja algunas funcionalidades sin implementar, debido al enfoque ágil de la metodología, el enfoque es, por ahora sólo en lo estrictamente necesario.

Esta primera versión contempla, la creación de campañas mediante un contrato intermedio, llamado 'CampaignFactory' el cual se encarga de generar instancias de las campañas, la razón del uso de este enfoque, es que desde el cliente web se llame a este primer contrato que se encargue de construir las instancias de 'Campaign', que representan las campañas.

```
1  //Version de Solidity a utilizar
2  pragma solidity  ^0.4.17;
3
4  contract CampaignFactory{
5    //Unico atributo del contrato generador de campanas el presente atributo se
       utiliza para devolver todas las direcciones instancias generadas
6      address[] public deployedCampaigns;
7
8    //Funcion que genera instancias del contrato Campaign, recibe un monto
       minimo de contribucion
9      function createCampaign(uint minimum) public{
10         address newCampaign = new Campaign(minimum, msg.sender);
11
12         //Una vez creada la instancia, se agrega al arreglo de direcciones de
       instancias de Campaigns
13         deployedCampaigns.push(newCampaign);
14     }
15
16   //Funcion que retorna un arreglo de direcciones con todas las intancias de
       Campaigns
17     function getDeployedCampaigns()public view returns (address[]){
18         return deployedCampaigns;
19     }
```

```
20
21 }
22
23 //Contrato principal, en este contrato se implementa toda la logica y
       funcionalidad de las campanas
24 contract Campaign{
25
26   //Estructura Request que representa una solicitud creada por el gerente del
       contrato
27     struct Request{
28
29       //String que representa la descripcion de la solicitud
30         string description;
31
32         //Entero positivo que representa el valor que desea retirar la
       solicitud de la campana
33         uint value;
34
35         //Direccion que recibira los fondos de la solicitud
36         address recipient;
37
38         //Variable booleana que representa si la solicitud ha sido completada
       o no
39         bool complete;
40
41         //Entero positivo que representa la cantidad de votos aprobados de la
       solicitud
42         uint approvalCount;
43
44         //Tipo de dato mapping, que realiza la correspondencia de una
       direccion con un valor booleano, en cuanto se realice una votacion, el
       mapeo de la direccion de esta variable sera asignado a True
45         mapping (address => bool) approvals;
46     }
47
48   //Lista de atributos del contrato
49
50     //Arreglo de solicitudes que posee el contrato
51     Request[] public requests;
52
53     //Direccion del gerente del contrato
54     address public manager;
55
56     //Contribucion minima
57     uint public minimumContribution;
58
59     //Mapeo de direcciones a valores booleanos que representan la lista de
       contribuyentes votantes
```

```
60    mapping (address => bool) public approvers;
61
62    //Contador de votantes
63    uint public approversCount;
64
65
66  //Funcion que restringe la posibilidad de una funcion a ser llamada, en este
       caso, retringe a la funcion a ser llamada por alguien que no sea el
      gerente de la campana
67    modifier restricted(){
68        require(msg.sender == manager);
69        _;
70    }
71
72  //Constructor del contrato, recibe sus parametros desde la funcion
      createCampaign de CampaignFactory
73    function Campaign(uint minimum, address creator) public{
74        manager = creator;
75        minimumContribution = minimum;
76    }
77
78  //Funcion que permite a una direccion contribuir a la campana, la palabra
      reservada 'payable' indica que la funcion debe recibir un valor en wei, el
       cual sera agregado a la instancia del contrato
79    function contribute() public payable{
80      //El valor con el que se llama a la funcion debe ser mayor a la minima
      contribucion establecida
81        require (msg.value > minimumContribution);
82
83    //Se mapea su valor a true, y se aumenta el contador de votantes
84        approvers[msg.sender] = true;
85        approversCount++;
86
87    }
88
89  //Funcion que crea una solicitud, recibe una descripcion, un valor y una
      direccion destino (Notese el atributo restricted: solo puede ser llamada
      por el gerente de la campana)
90    function createRequest(string description, uint value, address recipient)
      public restricted {
91
92      //Se crea la nueva estructura con los parametros recibidos
93        Request memory newRequest = Request({
94           description: description,
95           value: value,
96           recipient: recipient,
97           //Se inicializa en falso, puesto que se acaba de crear dicha
      solicitud y aun no se ha completado
```

```
98            complete: false,
99             //Se inicializa el contador de aprobados en cero.
100           approvalCount: 0
101       });
102
103    //Se agrega la solicitud recien creada al arreglo de solicitudes del
    contrato
104       requests.push(newRequest);
105
106    }
107
108   //Funcion llamada por los contribuyentes de la campana, para aprobar una
     solicitud, recibe un indice del arreglo de solicitudes para conocer cual
     solicitud desea votar para su aprobacion
109    function approveRequest(uint index) public{
110
111      //Localizar la solicitud segun el indice
112       Request storage request = requests[index];
113
114    //Para que la funcion continue su curso natural, es requerido que la
    direccion que llama la funcion sea un contribyente
115       require(approvers[msg.sender]);
116
117       //Tambien es necesario que la direccion que llama a la funcion NO haya
     votado anteriormente
118       require(!request.approvals[msg.sender]);
119
120    //La direccion aprobo la solicitud
121       request.approvals[msg.sender] = true;
122
123       //Se incrementa la cuenta de votos
124       request.approvalCount++;
125
126    }
127
128   //Funcion llamada para dar por finalizada una solicitud, recibe un indice
     para identificar la solicitud a finalizar, notese el atributo restricted,
     solo puede ser llamada por el gerente
129    function finalizeRequest(uint index) public restricted {
130      //Identificando la solicitud
131       Request storage request = requests[index];
132       //Es requerido que hayan votado positivamente la mitad de los
    contribuyentes
133       require(request.approvalCount > (approversCount / 2));
134
135       //Es requerido, ademas, que la solicitud no haya sido marcada como
    completada
136       require(!request.complete);
```

```
137
138     //Se transfieren los fondos a la direccion destino
139         request.recipient.transfer(request.value);
140
141         //Se marca la solicitud como completada
142         request.complete = true;
143     }
144 }
```

Algoritmo 6.1: Versión 0.1 de solución para el caso de uso

Algunos detalles que se deben hacer notar en esta primera versión del código son:

- No está contemplado la parametrización por parte del gerente de la campaña.
- Hacen falta la mitad de los votos de los contribuyentes para aprobar una solicitud
- No está implementada la funcionalidad de rechazar una solicitud

6.3.1.1. Prueba 0.1

Debido a este temprano enfoque para una solución, las pruebas se realizaron de manera manual mediante la herramienta Remix

Tipo de prueba	Funcional
Entorno de ejecución	Navegador Web Chrome
Ejecutante	Desarrollador de la herramienta
Herramienta usada	Remix
Veces ejecutada	1
Resultados	100 % de éxito

Tabla 6-1: Especificaciones de la prueba para la versión 0.1

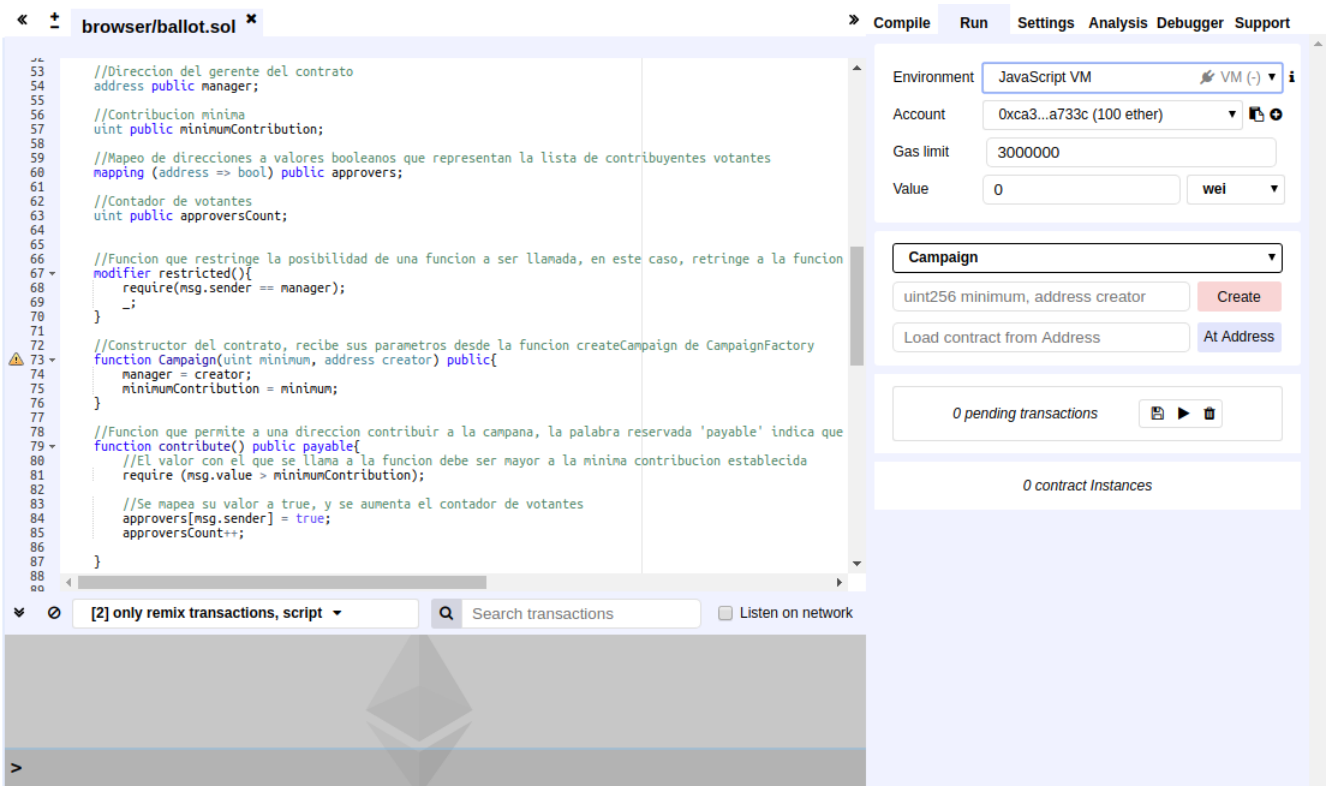

Figura 6-9: IDE de Remix, compilador de Solidity en en navegador web.

Sin embargo, en iteraciones futuras se contempló la implementación de pruebas de manera automática para asegurar que el código producido cumpla con los requerimientos solicitados.

6.3.2. Iteración 1: Compilación y primera prueba automática

La presente iteración presenta la compilación y despliegue del contrato en redes locales para realizar pruebas automáticas que garanticen su correcto funcionamiento.

Para lograr esta tarea, se maneja la primera versión del contrato contruída en la iteración 0, luego, se contruye el siguiente código que se encarga de crear, a partir del contrato creado, el ABI y *Bytecode* del contrato:

```
//Dependencias necesarias
const path = require('path');
const solc = require('solc');
const fs = require('fs-extra');

//Contruir la ruta donde se guardaran el ABI y el Bytecode del contrato compilado
const buildPath = path.resolve(__dirname, "build");

//Se vacia el contenido del directorio
fs.removeSync(buildPath);

//Se obtiene la ruta del contrato
const campaignPath = path.resolve(__dirname, 'contracts', 'Campaign.sol');

//Lectura del directorio dada su ruta
const source = fs.readFileSync(campaignPath, 'utf8');

//Compilacion, se utiliza la biblioteca 'solc'
const output = solc.compile(source, 1).contracts;

//Verificacion de que el directorio existe, si no existe, es creado
fs.ensureDirSync(buildPath);

//Iteracion sobre 'output', se crea un elemento JSON por cada objeto del 'output'.
for( let contract in output){
  fs.outputJsonSync(
    path.resolve(buildPath, contract.replace(':', '') + '.json'),
    output[contract]
  );
}
```

Algoritmo 6.2: Código que genera el ABI y *Bytecode* de un contrato dado

El código anterior crea dos objetos JSON, los cuales corresponden a los dos contratos que son parte del contrato global, ambos objetos JSON contienen el ABI y *Bytecode* de ambos contratos, a continuación se definen ambos términos:

- Aplicación de Interfaz binaria (ABI Application Binary Interface, en inglés): interfaz que es integrada con Web3 con la finalidad de que sea interpretado por JavaScript para poder interactuar con las funciones establecidas en los contratos mediante dicho lenguaje.
- Bytecode: Es la traducción del lenguaje Solidity a instrucciones que pueda entender la máquina virtual de Ethereum, es el archivo que se enviará a la cadena de bloques de Ethereum y a su máquina virtual.

Una vez que ambos archivos han sido generados, se procede a construir el código que realizará las pruebas automáticas.

Para construir las pruebas, se utiliza la biblioteca Mocha descrita en las tecnologías a utilizar, también se utilizan instancias de Web3 y Ganache para simular la red local de Ethereum sobre la cual se realizan las pruebas.

```
1  //Dependencias necesarias
2  const assert = require('assert');
3
4  //Ganache permite crear nuestro propio ambiente Ethereum localmente
5  const ganache = require('ganache-cli');
6  const Web3 = require('web3');
7  const web3 = new Web3(ganache.provider());
8
9  //Contratos a utilizar
10 const compiledFactory =  require('../ethereum/build/CampaignFactory.json');
11 const compiledCampaign =  require('../ethereum/build/Campaign.json');
12
13 //Lista de variables necesarias para el despligue
14 let accounts;
15 let factory;
16 let campaignAddress;
17 let campaign;
18
19 //Metodo llamado antes de realizar cada una de las pruebas, el metodo crea
      campanas automaticamente para realizar pruebas
20 beforeEach(async () => {
21
22   //Se obtiene una cuenta que hara el despliegue del contrato 'CampaignFactory
       '
23   accounts = await web3.eth.getAccounts();
24
```

```
25   //Se crea la instancia del contrato 'CampaignFactory'
26   factory = await new web3.eth.Contract(JSON.parse(compiledFactory.interface))
       .
27     deploy({ data: compiledFactory.bytecode }).
28     send({ from: accounts[0], gas: '2000000' });
29
30   //Se crea una campana utilizando la instancia recien creada
31   await factory.methods.createCampaign('100','0','0').send({
32     from: accounts[0],
33     gas: '2000000'
34   });
35
36   //Se obtienen las direcciones de todas las campanas creadas
37   const addresses = await factory.methods.getDeployedCampaigns().call();
38   campaignAddress = addresses[0];
39
40   //Se crea una instancia del contrato a partir de la direccion
41   campaign = await new web3.eth.Contract(JSON.parse(compiledCampaign.interface
       ), campaignAddress);
42
43 });
44
45 //Descripcion de las pruebas
46 describe('Pruebas a campanas', () => {
47
48   //Prueba que Instancia una CampaignFactory y una Campana
49   it('Instancia una CampaignFactory y una Campana', () => {
50
51     //Se verifica que ambas direcciones no sean NULL, asegurando su correcto
       despligue en la red local
52     assert.ok(factory.options.address);
53     assert.ok(campaign.options.address);
54   });
55
56   //Prueba que Marca al que llama a la campana como gerente
57   it('Marca al que llama a la campana como gerente', async () => {
58
59     //Se llama al atributo 'manager' de la instancia del contrato y se compara
       con la direccion actual que presenta la red local
60     const manager = await campaign.methods.manager().call();
61     assert.equal( accounts[0],manager);
62   });
63
64   //Prueba que Permite a los usuarios contribuir y marcarlos como
       contribuyentes
65   it('Permite a los usuarios contribuir y marcarlos como contribuyentes',
       async () => {
66
```

```
67    //Se realiza una contribucion a la campana
68    await campaign.methods.contribute().send({
69      value: '200',
70      from: accounts[1]
71    });
72
73    //Se llama al atributo approvers con la direccion que acaba de realizar la
      contribucion
74    const isContributor = await campaign.methods.approvers(accounts[1]).call()
      ;
75
76    //El valor debe ser true
77    assert(isContributor);
78
79  });
80
81  //Prueba que asegura de requerir una contribucion minima
82  it('Asegura de requerir una contribucion minima', async () => {
83
84    //Se realizar una contribucion menor al monto minimo establecido (100)
85    try {
86      await campaign.methods.contribute().send({
87        value:'5',
88        from: accounts[1]
89      });
90      //Si se llega a este punto de la prueba, es que algo malo ocurrio, la
      idea es que el catch se ejecute
91      assert(false);
92    } catch (err) {
93
94      //Si el catch se ejecuta, hubo un fallo, lo cual es el comportamiento
      esperado
95      assert(err);
96    }
97  });
98
99  //Prueba que permite al gerente crear solicitudes
100 it('Permite al gerente crear solicitudes', async () => {
101
102   //Se llama al metodo 'createRequest'
103   await campaign.methods.createRequest('Comprar baterias', '100', accounts
      [1]).send({
104     from: accounts[0],
105     gas: '1000000'
106   });
107
108   //Se obtiene la solicitud recien creada
109   const request = await campaign.methods.requests(0).call();
```

```
110
111     //Se verifica que la descripcion de la solicitud sea 'Comprar baterias'
112     assert.equal('Comprar baterias', request.description);
113   });
114
115   //Prueba que permite procesar solicitudes
116   it('Procesar solicitudes', async () => {
117
118     //Primero se realiza una contribucion
119     await campaign.methods.contribute().send({
120       from: accounts[0],
121       value: web3.utils.toWei('10', 'ether')
122     });
123
124     //Se crea una solicitud
125     await campaign.methods.createRequest('A', web3.utils.toWei('5', 'ether'),
    accounts[1]).send({
126       from:accounts[0],
127       gas: '1000000'
128     });
129
130     //Se aprueba la solicitud recien creada
131     await campaign.methods.approveRequest(0).send({
132       from:accounts[0],
133       gas: '1000000'
134     });
135
136     //Se finaliza la solicitud recien aprobada
137     await campaign.methods.finalizeRequest(0).send({
138       from:accounts[0],
139       gas:'1000000'
140     });
141
142     //Se obtiene el balance de la cuenta destino
143     let balance = await web3.eth.getBalance(accounts[1]);
144
145     //El balance se pasa de wei a ether
146     balance = web3.utils.fromWei(balance, 'ether');
147     balance = parseFloat(balance);
148
149     console.log(balance);
150
151     //Se verifica que el balance sea mayor a lo que tenia en un principio
152     assert(balance > 104);
153   });
```

```
154 });
```

Algoritmo 6.3: Código que realiza pruebas automáticas de despliegue de un contrato y funcionalidades

Tipo de prueba	Funcional
Entorno de ejecución	Ubuntu 16.04 LTS
Ejecutante	Desarrollador de la herramienta
Herramienta usada	Mocha
Veces ejecutada	1
Resultados	100 % de los requerimientos estipulados

Tabla 6-2: Especificaciones de la prueba automática para la versión 0.1

Es importante acotar la información con respecto al equipo de hardware utilizado para realizar dicha prueba, el cual es:

Componente	Especificación
Sistema Operativo	Ubuntu 16.04 LTS
Memoria	2GB
Procesador	Intel Core i5-3470 CPU @ 3.20GHz x 4
Disco Duro	500GB

Tabla 6-3: Especificaciones de hardware para pruebas en Mocha

Una vez definidas las pruebas para la campaña, se hace uso del comando 'npm run test', provisto por Mocha en el terminal para ejecutar el código anterior y que se realicen las pruebas de manera automática:

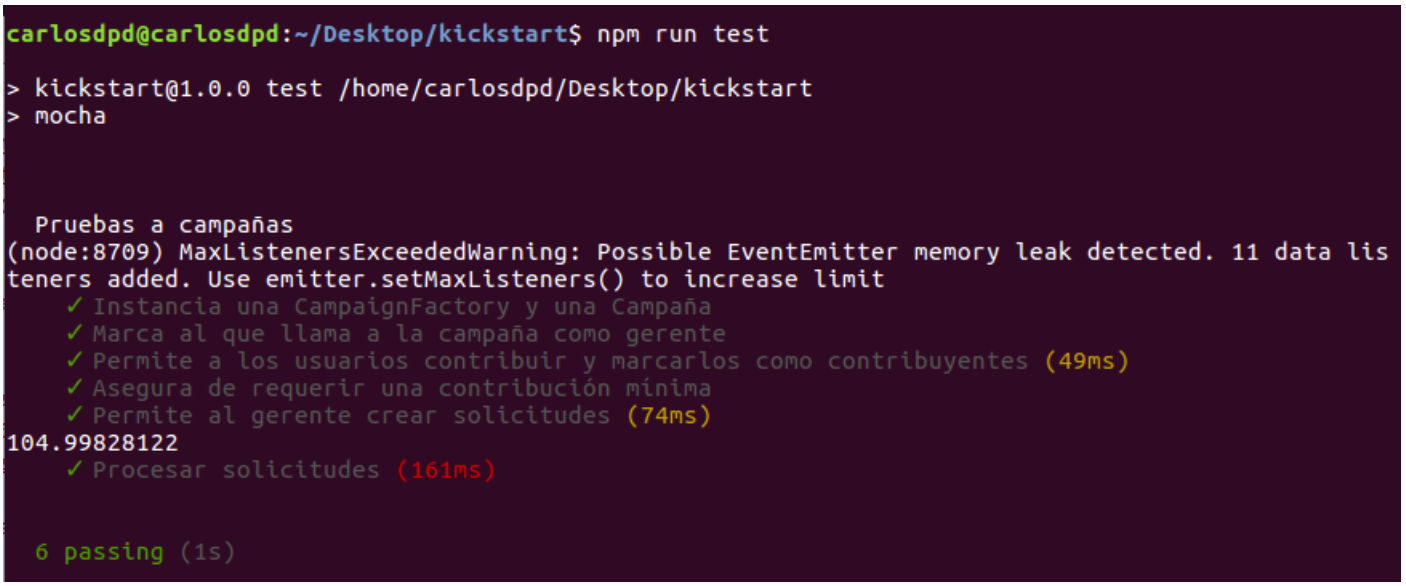

Figura 6-10: Ejecución de pruebas automáticas mediante el uso de la biblioteca Mocha.

Como se pude observar en la Figura **6-10**, las pruebas se realizaron de manera exitosa, se logró hacer el despligue del ABI y el *Bytecode* en una red local Ethereum y se validó el correcto funcionamiento.

6.3.3. Iteración 2: Segundo acercamiento, reestructuración de pruebas

En vista de las pruebas exitosas con respecto a un primer prototipo de contrato construido, en la presente iteración se tomarán en cuenta algunos aspectos dejados de lado en la iteración 0.

El siguiente código representa una siguiente versión del contrato el cual maneja una configuración más amplia de parte del gerente a la hora de definir los parámetros de su instancia de campaña:

```
1  //Version de Solidity a utilizar
2  pragma solidity  ^0.4.17;
3
4  contract CampaignFactory{
5    //Unico atributo del contrato generador de campanas el presente atributo se
       utiliza para devolver todas las direcciones instancias generadas
6      address[] public deployedCampaigns;
7
8    //Funcion que genera instancias del contrato Campaign, recibe un monto
       minimo de contribucion, un monto maximo de contribucion y un numero maximo
        de contribuyentes
9      function createCampaign(uint minimum, uint maximum, uint maxCont,) public{
10         address newCampaign = new Campaign(minimum, maximum, maxCont, msg.
       sender);
11
12         //Una vez creada la instancia, se agrega al arreglo de direcciones de
       instancias de Campaigns
13         deployedCampaigns.push(newCampaign);
14     }
15
16   //Funcion que retorna un arreglo de direcciones con todas las intancias de
       Campaigns
17     function getDeployedCampaigns()public view returns (address[]){
18         return deployedCampaigns;
19     }
20
21 }
22
23 //Contrato principal, en este contrato se implementa toda la logica y
       funcionalidad de las campanas
24 contract Campaign{
25
```

```
26  //Estructura Request que representa una solicitud creada por el gerente del
    contrato
27    struct Request{
28
29      //String que representa la descripcion de la solicitud
30        string description;
31
32        //Entero positivo que representa el valor que desea retirar la
    solicitud de la campana
33        uint value;
34
35        //Direccion que recibira los fondos de la solicitud
36        address recipient;
37
38        //Variable booleana que representa si la solicitud ha sido completada
    o no
39        bool complete;
40
41        //Entero positivo que representa la cantidad de votos aprobados de la
    solicitud
42        uint approvalCount;
43
44        //Tipo de dato mapping, que realiza la correspondencia de una
    direccion con un valor booleano, en cuanto se realice una votacion, el
    mapeo de la direccion de esta variable sera asignado a True
45        mapping (address => bool) approvals;
46    }
47
48  //Lista de atributos del contrato
49
50    //Arreglo de solicitudes que posee el contrato
51    Request[] public requests;
52
53    //Direccion del gerente del contrato
54    address public manager;
55
56    //Contribucion minima
57    uint public minimumContribution;
58
59    //Contribucion maxima
60    uint public maximumContribution;
61
62    //Numero maximo de contribuyentes
63    uint public maxContributors;
64
65    //Mapeo de direcciones a valores booleanos que representan la lista de
    contribuyentes votantes
66    mapping (address => bool) public approvers;
```

```
67
68      //Contador de votantes
69      uint public approversCount;
70
71
72  //Funcion que restringe la posibilidad de una funcion a ser llamada, en este
        caso, retringe a la funcion a ser llamada por alguien que no sea el
      gerente de la campana
73      modifier restricted(){
74          require(msg.sender == manager);
75          _;
76      }
77
78  //Constructor del contrato, recibe sus parametros desde la funcion
      createCampaign de CampaignFactory
79      function Campaign(uint minimum, address creator) public{
80
81        //Es requerido que el monto minimo de aprobacion sea mayor a cero
82        require (minimum > 0);
83          manager = creator;
84          minimumContribution = minimum;
85          maximumContribution =  maximum;
86          maxContributors = maxCont;
87      }
88
89  //Funcion que permite a una direccion contribuir a la campana, la palabra
      reservada 'payable' indica que la funcion debe recibir un valor en wei, el
        cual sera agregado a la instancia del contrato
90      function contribute() public payable{
91        //Requerimientos basicos para que la funcion pueda seguir su curso
      natural:
92          //-El valor con el que se llama a la funcion debe ser mayor a la
      minima contribucion establecida
93          //-El valor con el que se llama a la funcion debe ser menor a la
      maxima contribucion establecida, o ser cero, para que no haya limite en la
        contribucion
94          //-El numero de direcciones que aprueban una solicitud no puede ser
      mayor al numero maximo de contribuyentes, o debe ser 0, en caso de que no
      haya limite
95          require (msg.value > minimumContribution);
96          require (msg.value > minimumContribution && (msg.value <
      maximumContribution || maximumContribution == 0 )&& (approversCount <
      maxContributors || maxContributors == 0));
97
98
99          //Este condicional permite a una direccion contribuir varias veces a
      una misma campama
100         if( approvers[msg.sender] == false){
```

```
101
102         //Se mapea su valor a true, y se aumenta el contador de votantes
103         approvers[msg.sender] = true;
104         approversCount++;
105       }
106
107   }
108
109 //Funcion que crea una solicitud, recibe una descripcion, un valor y una direccion destino (Notese el atributo restricted: solo puede ser llamada por el gerente de la campana)
110   function createRequest(string description, uint value, address recipient) public restricted {
111
112     //Se crea la nueva estructura con los parametros recibidos
113       Request memory newRequest = Request({
114          description: description,
115          value: value,
116          recipient: recipient,
117          //Se inicializa en falso, puesto que se acaba de crear dicha solicitud y aun no se ha completado
118          complete: false,
119           //Se inicializa el contador de aprobados en cero.
120          approvalCount: 0
121       });
122
123   //Se agrega la solicitud recien creada al arreglo de solicitudes del contrato
124       requests.push(newRequest);
125
126   }
127
128 //Funcion llamada por los contribuyentes de la campana, para aprobar una solicitud, recibe un indice del arreglo de solicitudes para conocer cual solicitud desea votar para su aprobacion
129   function approveRequest(uint index) public{
130
131     //Localizar la solicitud segun el indice
132       Request storage request = requests[index];
133
134   //Para que la funcion continue su curso natural, es requerido que la direccion que llama la funcion sea un contribyente
135       require(approvers[msg.sender]);
136
137       //Tambien es necesario que la direccion que llama a la funcion NO haya votado anteriormente
138       require(!request.approvals[msg.sender]);
139
```

```
140     //La direccion aprobo la solicitud
141         request.approvals[msg.sender] = true;
142 
143         //Se incrementa la cuenta de votos
144         request.approvalCount++;
145 
146     }
147 
148   //Funcion llamada para dar por finalizada una solicitud, recibe un indice
      para identificar la solicitud a finalizar, notese el atributo restricted,
      solo puede ser llamada por el gerente
149     function finalizeRequest(uint index) public restricted {
150       //Identificando la solicitud
151         Request storage request = requests[index];
152         //Es requerido que hayan votado positivamente la mitad de los
      contribuyentes
153         require(request.approvalCount > (approversCount / 2));
154 
155         //Es requerido, ademas, que la solicitud no haya sido marcada como
      completada
156         require(!request.complete);
157 
158     //Se transfieren los fondos a la direccion destino
159         request.recipient.transfer(request.value);
160 
161         //Se marca la solicitud como completada
162         request.complete = true;
163     }
164 
165     //Funcion que retorna un resumen de la campana
166     function getSummary() public view returns(uint, uint, uint, uint, uint,
      uint, address){
167 
168       //Retornar todos los atributos de la campana
169         return (
170           minimumContribution,
171           maximumContribution,
172           maxContributors,
173           this.balance,
174           requests.length,
175           approversCount,
176           manager
177           );
178     }
179 
180   //Funcion que retorna el numero de solicitudes
181     function getRequestCount() public view returns(uint){
182         return requests.length;
```

```
183     }
184 }
```

Algoritmo 6.4: Versión 0.2 de solución para el caso de uso

La versión 0.2 de la solución del caso de uso presenta las siguientes evoluciones:

- Permite al gerente ingresar una contribución máxima.
- Permite al gerente ingresar un número máximo de contribuyentes.
- Valida que la contribución mínima debe ser mayor a 0.
- Si la contribución máxima o el número de contribuyentes es 0, se toma como si no hubiera límite.
- Se puede obtener un objeto que contiene el resumen de una campaña.
- Se puede hacer una llamada al contrato para conocer el número de solicitudes.

Muchas de estas nuevas adiciones son útiles a la hora de implementar el cliente web, puesto que mientras más detalles se puedan mostrar de una campaña, se mostrará más confianza al usuario final a la hora de contribuir a una campaña.

6.3.3.1. Prueba versión 0.2

Una vez construida la evolución de un primer contrato, también debe modificarse el código que realiza las pruebas, puesto que ahora, el constructor recibe nuevos parámetros y existen nuevas validaciones que podrían ocasionar errores en el comportamiento esperado.

Tipo de prueba	Funcional
Entorno de ejecución	MAc OS Sierra
Ejecutante	Desarrollador de la herramienta
Herramienta usada	Mocha
Veces ejecutada	5
Resultados	100 % de los requerimientos estipulados

Tabla 6-4: Especificaciones de la prueba para la versión 0.2

Para esta prueba, se realizó el despliegue mediante otro equipo de hardware con la finalidad de descartar el funcionamiento exclusivo de la prueba en un sistema operativo determinado, el nuevo hardware que realiza la prueba se describe a continuación:

Componente	Especificación
Sistema Operativo	Mac OS Sierra
Memoria	4GB
Procesador	Mac Mini 2.3 GHz Core i5 (I5-2415M)
Disco Duro	1TB

Tabla 6-5: Especificaciones de hardware para pruebas en Mocha. Ambiente MacOS

Se presenta el código que realiza las pruebas anteriores, asegurando que todas las funcionalidades anteriores siguen cumpliendo sus tareas, y además se agrega una nueva prueba de funcionamiento.

```
1  //Dependencias necesarias
2  const assert = require('assert');
3
4  //Ganache permite crear nuestro propio ambiente Ethereum localmente
5  const ganache = require('ganache-cli');
6  const Web3 = require('web3');
7  const web3 = new Web3(ganache.provider());
8
9  //Contratos a utilizar
10 const compiledFactory =  require('../ethereum/build/CampaignFactory.json');
11 const compiledCampaign =  require('../ethereum/build/Campaign.json');
12
13 //Lista de variables necesarias para el despligue
14 let accounts;
15 let factory;
16 let campaignAddress;
17 let campaign;
18
19 //Metodo llamado antes de realizar cada una de las pruebas, el metodo crea
      campanas automaticamente para realizar pruebas
20 beforeEach(async () => {
21
22   //Se obtiene una cuenta que hara el despliegue del contrato 'CampaignFactory
       '
23   accounts = await web3.eth.getAccounts();
24
25   //Se crea la instancia del contrato 'CampaignFactory'
26   factory = await new web3.eth.Contract(JSON.parse(compiledFactory.interface))
       .
27     deploy({ data: compiledFactory.bytecode }).
28     send({ from: accounts[0], gas: '2000000' });
29
30   //Se crea una campana utilizando la instancia recien creada
31   await factory.methods.createCampaign('100',web3.utils.toWei('12', 'ether'),'
       0','50','50').send({
32     from: accounts[0],
33     gas: '2000000'
34   });
35
36   //Se obtienen las direcciones de todas las campanas creadas
37   const addresses = await factory.methods.getDeployedCampaigns().call();
38   campaignAddress = addresses[0];
39
40   //Se crea una instancia del contrato a partir de la direccion
```

```
41   campaign = await new web3.eth.Contract(JSON.parse(compiledCampaign.interface
       ), campaignAddress);
42
43 });
44
45 //Descripcion de las pruebas
46 describe('Pruebas a campanas', () => {
47
48   //Prueba que Instancia una CampaignFactory y una Campana
49   it('Instancia una CampaignFactory y una Campana', () => {
50
51     //Se verifica que ambas direcciones no sean NULL, asegurando su correcto
      despligue en la red local
52     assert.ok(factory.options.address);
53     assert.ok(campaign.options.address);
54   });
55
56   //Prueba que Marca al que llama a la campana como gerente
57   it('Marca al que llama a la campana como gerente', async () => {
58
59     //Se llama al atributo 'manager' de la instancia del contrato y se compara
      con la direccion actual que presenta la red local
60     const manager = await campaign.methods.manager().call();
61     assert.equal( accounts[0],manager);
62   });
63
64   //Prueba que Permite a los usuarios contribuir y marcarlos como
      contribuyentes
65   it('Permite a los usuarios contribuir y marcarlos como contribuyentes',
      async () => {
66
67     //Se realiza una contribucion a la campana
68     await campaign.methods.contribute().send({
69       value: '200',
70       from: accounts[1]
71     });
72
73     //Se llama al atributo approvers con la direccion que acaba de realizar la
      contribucion
74     const isContributor = await campaign.methods.approvers(accounts[1]).call()
      ;
75
76     //El valor debe ser true
77     assert(isContributor);
78
79   });
80
81   //Prueba que asegura de requerir una contribucion minima
```

```
82  it('Asegura de requerir una contribucion minima', async () => {
83
84    //Se realizar una contribucion menor al monto minimo establecido (100)
85    try {
86      await campaign.methods.contribute().send({
87        value:'5',
88        from: accounts[1]
89      });
90      //Si se llega a este punto de la prueba, es que algo malo ocurrio, la
      idea es que el catch se ejecute
91      assert(false);
92    } catch (err) {
93
94      //Si el catch se ejecuta, hubo un fallo, lo cual es el comportamiento
      esperado
95      assert(err);
96    }
97  });
98
99  //Prueba que asegura que una contribucion minima no puede ser 0
100 it('Asegura que una contribucion minima no puede ser 0', async () => {
101
102   //Se realizar una contribucion de 0
103   try {
104     await campaign.methods.contribute().send({
105       value:'0',
106       from: accounts[1]
107     });
108     //Si se llega a este punto de la prueba, es que algo malo ocurrio, la
      idea es que el catch se ejecute
109     assert(false);
110   } catch (err) {
111
112     //Si el catch se ejecuta, hubo un fallo, lo cual es el comportamiento
      esperado
113     assert(err);
114   }
115 });
116
117 //Prueba que asegura de no sobrepasar la contribucion maxima
118   it('Asegura de no sobrepasar la contribucion maxima', async () => {
119
120     //Se realizar una contribucion mayor al monto maximo establecido (10
      ether)
121     try {
122       await campaign.methods.contribute().send({
123         value:web3.utils.toWei('15', 'ether'),
124         from: accounts[1]
```

```
125         });
126         //Si se llega a este punto de la prueba, es que algo malo ocurrio, la
      idea es que el catch se ejecute
127         assert(false);
128       } catch (err) {
129
130         //Si el catch se ejecuta, hubo un fallo, lo cual es el comportamiento
      esperado
131         assert(err);
132       }
133     });
134
135   //Prueba que permite al gerente crear solicitudes
136   it('Permite al gerente crear solicitudes', async () => {
137
138     //Se llama al metodo 'createRequest'
139     await campaign.methods.createRequest('Comprar baterias', '100', accounts
      [1]).send({
140       from: accounts[0],
141       gas: '1000000'
142     });
143
144     //Se obtiene la solicitud recien creada
145     const request = await campaign.methods.requests(0).call();
146
147     //Se verifica que la descripcion de la solicitud sea 'Comprar baterias'
148     assert.equal('Comprar baterias', request.description);
149   });
150
151   //Prueba que permite procesar solicitudes
152   it('Procesar solicitudes', async () => {
153
154     //Primero se realiza una contribucion
155     await campaign.methods.contribute().send({
156       from: accounts[0],
157       value: web3.utils.toWei('10', 'ether')
158     });
159
160     //Se crea una solicitud
161     await campaign.methods.createRequest('A', web3.utils.toWei('5', 'ether'),
      accounts[1]).send({
162       from:accounts[0],
163       gas: '1000000'
164     });
165
166     //Se aprueba la solicitud recien creada
167     await campaign.methods.approveRequest(0).send({
168       from:accounts[0],
```

```
169       gas: '1000000'
170     });
171
172     //Se finaliza la solicitud recien aprobada
173     await campaign.methods.finalizeRequest(0).send({
174       from:accounts[0],
175       gas:'1000000'
176     });
177
178     //Se obtiene el balance de la cuenta destino
179     let balance = await web3.eth.getBalance(accounts[1]);
180
181     //El balance se pasa de wei a ether
182     balance = web3.utils.fromWei(balance, 'ether');
183     balance = parseFloat(balance);
184
185     console.log(balance);
186
187     //Se verifica que el balance sea mayor a lo que tenia en un principio
188     assert(balance > 104);
189
190   });
191
192 });
```

Algoritmo 6.5: Código que realiza pruebas automáticas de despliegue de un contrato y funcionalidades. Versión 0.2

La prueba ahora contempla el hecho de que existe una contribución máxima, y que no una contribución no debe sobrepasar ese monto. El comando que ejecuta las pruebas vuelve a ejecutarse para realizar dichas pruebas modificadas:

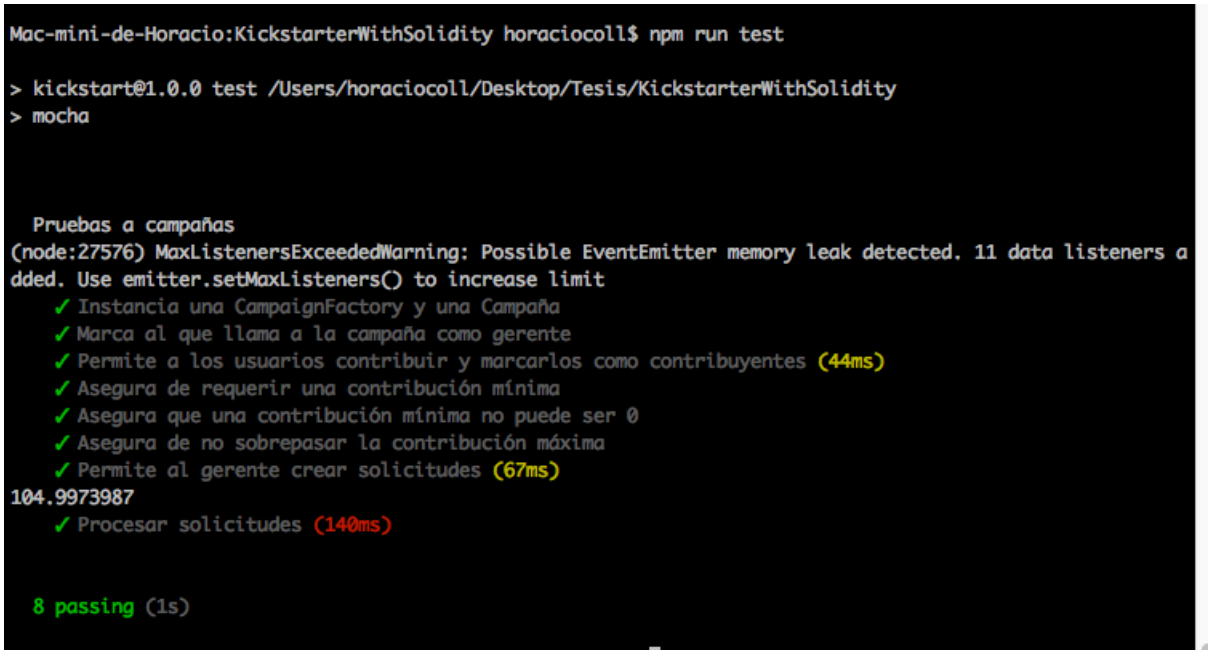

Figura 6-11: Ejecución de pruebas automáticas mediante el uso de la biblioteca Mocha para la versioón 0.2 de la solución.

Sin embargo, aún quedan piezas importantes de la problemática por resolver:

- Lógica para rechazar una solicitud.
- Tasas de aprobación y rechazo de una solicitud.
- Expiración de una solicitud.

6.3.4. Iteración 3: Rechazo de solicitudes y tasas

Esta iteración presenta la implementación del rechazo de solicitudes así como también los atributos de tasa de rechazo y tasa de aprobación de una solicitud. Esto representa un nivel más alto aún de personalización de la campaña.

Este tipo de adiciones al código lo hacen mucho más versátil y manejable tanto como para el gerente de la campaña, como a los usuarios a la hora de contribuir a una campaña, mientras más información se les pueda brindar, más confianza tendrán y serán más propensos a realizar contribuciones.
A continuación se presenta la versión 0.3 de la solución:

```
1  //Version de Solidity a utilizar
2  pragma solidity  ^0.4.17;
3
4  contract CampaignFactory{
5    //Unico atributo del contrato generador de campanas el presente atributo se
       utiliza para devolver todas las direcciones instancias generadas
6      address[] public deployedCampaigns;
7
8    //Funcion que genera instancias del contrato Campaign, recibe un monto
       minimo de contribucion, un monto maximo de contribucion, una cantidad
       maxima de contribuyentes, una tasa de aprobacion y una tasa de rechazo
9      function createCampaign(uint minimum, uint maximum, uint maxCont, uint
       approveRate, uint rejectRate) public{
10         address newCampaign = new Campaign(minimum, maximum, maxCont, msg.
       sender, approveRate, rejectRate);
11
12         //Una vez creada la instancia, se agrega al arreglo de direcciones de
       instancias de Campaigns
13         deployedCampaigns.push(newCampaign);
14     }
15
16   //Funcion que retorna un arreglo de direcciones con todas las intancias de
       Campaigns
17     function getDeployedCampaigns()public view returns (address[]){
18         return deployedCampaigns;
19     }
20
21 }
22
23 //Contrato principal, en este contrato se implementa toda la logica y
       funcionalidad de las campanas
24 contract Campaign{
25
26   //Estructura Request que representa una solicitud creada por el gerente del
       contrato
27     struct Request{
28
29       //String que representa la descripcion de la solicitud
30         string description;
31
32         //Entero positivo que representa el valor que desea retirar la
       solicitud de la campana
33         uint value;
34
35         //Direccion que recibira los fondos de la solicitud
36         address recipient;
37
38         //Variable booleana que representa si la solicitud ha sido completada
```

```
    o no
39          bool complete;
40
41          //Entero positivo que representa la cantidad de votos aprobados de la
        solicitud
42          uint approvalCount;
43
44          //Entero positivo que representa la cantidad de votos rechazados de la
         solicitud
45          uint rejectsCount;
46
47          //Tipo de dato mapping, que realiza la correspondencia de una
        direccion con un valor booleano, en cuanto se realice una votacion, el
        mapeo de la direccion de esta variable sera asignado a True
48          mapping (address => bool) approvals;
49      }
50
51  //Lista de atributos del contrato
52
53      //Arreglo de solicitudes que posee el contrato
54      Request[] public requests;
55
56      //Direccion del gerente del contrato
57      address public manager;
58
59      //Contribucion minima
60      uint public minimumContribution;
61
62      //Contribucion maxima
63      uint public maximumContribution;
64
65      //Numero maximo de contribuyentes
66      uint public maxContributors;
67
68      //Porcentaje para el porcentaje de aprobacion
69      uint public approvalRate;
70
71      //Porcentaje para el porcentaje de rechazo
72      uint public rejectedRate;
73
74      //Mapeo de direcciones a valores booleanos que representan la lista de
        contribuyentes votantes
75      mapping (address => bool) public approvers;
76
77      //Contador de votantes
78      uint public approversCount;
79
80
```

```
81  //Funcion que restringe la posibilidad de una funcion a ser llamada, en este caso, retringe a la funcion a ser llamada por alguien que no sea el gerente de la campana
82   modifier restricted(){
83       require(msg.sender == manager);
84       _;
85   }
86
87  //Constructor del contrato, recibe sus parametros desde la funcion createCampaign de CampaignFactory
88   function Campaign(uint minimum, uint maximum, uint maxCont, address creator, uint approveRate, uint rejectRate) public{
89
90     //Es requerido que el monto minimo de aprobacion sea mayor a cero
91     require (minimum > 0);
92       manager = creator;
93       minimumContribution = minimum;
94       maximumContribution =  maximum;
95       maxContributors = maxCont;
96       approvalRate = approveRate;
97       rejectedRate = rejectRate;
98   }
99
100 //Funcion que permite a una direccion contribuir a la campana, la palabra reservada 'payable' indica que la funcion debe recibir un valor en wei, el cual sera agregado a la instancia del contrato
101  function contribute() public payable{
102    //Requerimientos basicos para que la funcion pueda seguir su curso natural:
103      //-El valor con el que se llama a la funcion debe ser mayor a la minima contribucion establecida
104      //-El valor con el que se llama a la funcion debe ser menor a la maxima contribucion establecida, o ser cero, para que no haya limite en la contribucion
105      //-El numero de direcciones que aprueban una solicitud no puede ser mayor al numero maximo de contribuyentes, o debe ser 0, en caso de que no haya limite
106      require (msg.value > minimumContribution);
107      require (msg.value > minimumContribution && (msg.value < maximumContribution || maximumContribution == 0 )&& (approversCount < maxContributors || maxContributors == 0));
108
109
110      //Este condicional permite a una direccion contribuir varias veces a una misma campama
111      if( approvers[msg.sender] == false){
112
113        //Se mapea su valor a true, y se aumenta el contador de votantes
```

```
114             approvers[msg.sender] = true;
115             approversCount++;
116         }
117
118     }
119
120 //Funcion que crea una solicitud, recibe una descripcion, un valor y una direccion destino (Notese el atributo restricted: solo puede ser llamada por el gerente de la campana)
121     function createRequest(string description, uint value, address recipient) public restricted {
122
123         //Se crea la nueva estructura con los parametros recibidos
124         Request memory newRequest = Request({
125             description: description,
126             value: value,
127             recipient: recipient,
128             //Se inicializa en falso, puesto que se acaba de crear dicha solicitud y aun no se ha completado
129             complete: false,
130              //Se inicializa el contador de aprobados en cero.
131             approvalCount: 0
132             //Se inicializa el contador de rechazados en cero.
133             rejectsCount: 0
134         });
135
136     //Se agrega la solicitud recien creada al arreglo de solicitudes del contrato
137         requests.push(newRequest);
138
139     }
140
141 //Funcion llamada por los contribuyentes de la campana, para aprobar una solicitud, recibe un indice del arreglo de solicitudes para conocer cual solicitud desea votar para su aprobacion
142     function approveRequest(uint index) public{
143
144         //Localizar la solicitud segun el indice
145         Request storage request = requests[index];
146
147     //Para que la funcion continue su curso natural, es requerido que la direccion que llama la funcion sea un contribyente
148         require(approvers[msg.sender]);
149
150         //Tambien es necesario que la direccion que llama a la funcion NO haya votado anteriormente
151         require(!request.approvals[msg.sender]);
152
```

```
153     //La direccion aprobo la solicitud
154         request.approvals[msg.sender] = true;
155 
156         //Se incrementa la cuenta de votos
157         request.approvalCount++;
158 
159     }
160 
161     //Funcion llamada por los contribuyentes de la campana, para rechazar una solicitud, recibe un indice del arreglo de solicitudes para conocer cual solicitud desea votar para su rechazo
162     function rejectRequest(uint index) public{
163 
164         //Localizar la solicitud segun el indice
165         Request storage request = requests[index];
166 
167         //Para que la funcion continue su curso natural, es requerido que la direccion que llama la funcion sea un contribyente
168         require(approvers[msg.sender]);
169 
170         //Tambien es necesario que la direccion que llama a la funcion NO haya votado anteriormente
171         require(!request.approvals[msg.sender]);
172 
173         //La direccion rechazo la solicitud
174         request.approvals[msg.sender] = true;
175 
176         //Se incrementa la cuenta de votos
177         request.rejectsCount++;
178 
179     }
180 
181   //Funcion llamada para dar por finalizada una solicitud, recibe un indice para identificar la solicitud a finalizar, notese el atributo restricted, solo puede ser llamada por el gerente
182     function finalizeRequest(uint index) public restricted {
183       //Identificando la solicitud
184         Request storage request = requests[index];
185 
186         //Es requerido, ademas, que la solicitud no haya sido marcada como completada
187         require(!request.complete);
188 
189     //Si es el caso de aprobacion, se traspasa el dinero
190         if(request.approvalCount > (approversCount*(approvalRate))/100){
191 
192             //Se transfieren los fondos a la direccion destino
193             request.recipient.transfer(request.value);
```

```
194
195             //Se marca la solicitud como completada
196             request.complete = true;
197         }
198
199         //En caso que haya sido rechazo, se finaliza la solicitud sin enviar
    fondos
200         if(request.rejectsCount > (approversCount*(rejectedRate))/100){
201
202             //Se marca la solicitud como completada
203             request.complete = true;
204         }
205     }
206
207     //Funcion que retorna un resumen de la campana
208     function getSummary() public view returns(uint, uint, uint, uint, uint,
    uint, uint, uint, address){
209
210       //Retornar todos los atributos de la campana
211         return (
212           minimumContribution,
213           maximumContribution,
214           maxContributors,
215           this.balance,
216           requests.length,
217           approversCount,
218           approvalRate,
219           rejectedRate,
220           manager
221           );
222     }
223
224   //Funcion que retorna el numero de solicitudes
225     function getRequestCount() public view returns(uint){
226         return requests.length;
227     }
228 }
```

Algoritmo 6.6: Versión 0.3 de solución para el caso de uso

Con las modificaciones realizadas al código de la solución, se implementó satisfactoriamente la lógica que permite rechazar solicitudes, definir tasas de aprobación y rechazo de solicitudes, esto agrega un valor importante a la versatilidad de la solución y se acerca cada vez más a cubrir todas las necesidades planteadas en el caso de uso.

6.3.4.1. Prueba versión 0.3

Las pruebas a realizar a esta nueva versión deben también ser modificadas para asegurar dos cosas, la primera, que funcionalidades anteriores sigan cumpliendo a cabalidad las funciones para las cuales fueron construídas, y segunda, asegurar la calidad y eficacia de las nuevas funcionalidades incorporadas, a continuación se presenta el código de prueba adaptado a la nueva versión:

Tipo de prueba	Funcional
Entorno de ejecución	Ubuntu Linux 16.04 LTS
Ejecutante	Desarrollador de la herramienta
Herramienta usada	Mocha
Veces ejecutada	3
Resultados	100 % de los requerimientos estipulados

Tabla 6-6: Especificaciones de la prueba para la versión 0.3

```
1  //Dependencias necesarias
2  const assert = require('assert');
3
4  //Ganache permite crear nuestro propio ambiente Ethereum localmente
5  const ganache = require('ganache-cli');
6  const Web3 = require('web3');
7  const web3 = new Web3(ganache.provider());
8
9  //Contratos a utilizar
10 const compiledFactory =  require('../ethereum/build/CampaignFactory.json');
11 const compiledCampaign =  require('../ethereum/build/Campaign.json');
12
13 //Lista de variables necesarias para el despligue
14 let accounts;
15 let factory;
16 let campaignAddress;
17 let campaign;
18
19 //Metodo llamado antes de realizar cada una de las pruebas, el metodo crea
      campanas automaticamente para realizar pruebas
20 beforeEach(async () => {
21
22   //Se obtiene una cuenta que hara el despliegue del contrato 'CampaignFactory
       '
23   accounts = await web3.eth.getAccounts();
24
25   //Se crea la instancia del contrato 'CampaignFactory'
26   factory = await new web3.eth.Contract(JSON.parse(compiledFactory.interface))
       .
27     deploy({ data: compiledFactory.bytecode }).
```

```
28      send({ from: accounts[0], gas: '2000000' });
29
30    //Se crea una campana utilizando la instancia recien creada
31    await factory.methods.createCampaign('100',web3.utils.toWei('12', 'ether'),'
        0','50','50').send({
32      from: accounts[0],
33      gas: '2000000'
34    });
35
36    //Se obtienen las direcciones de todas las campanas creadas
37    const addresses = await factory.methods.getDeployedCampaigns().call();
38    campaignAddress = addresses[0];
39
40    //Se crea una instancia del contrato a partir de la direccion
41    campaign = await new web3.eth.Contract(JSON.parse(compiledCampaign.interface
        ), campaignAddress);
42
43 });
44
45 //Descripcion de las pruebas
46 describe('Pruebas a campanas', () => {
47
48    //Prueba que Instancia una CampaignFactory y una Campana
49    it('Instancia una CampaignFactory y una Campana', () => {
50
51      //Se verifica que ambas direcciones no sean NULL, asegurando su correcto
        despligue en la red local
52      assert.ok(factory.options.address);
53      assert.ok(campaign.options.address);
54    });
55
56    //Prueba que Marca al que llama a la campana como gerente
57    it('Marca al que llama a la campana como gerente', async () => {
58
59      //Se llama al atributo 'manager' de la instancia del contrato y se compara
        con la direccion actual que presenta la red local
60      const manager = await campaign.methods.manager().call();
61      assert.equal( accounts[0],manager);
62    });
63
64    //Prueba que Permite a los usuarios contribuir y marcarlos como
        contribuyentes
65    it('Permite a los usuarios contribuir y marcarlos como contribuyentes',
        async () => {
66
67      //Se realiza una contribucion a la campana
68      await campaign.methods.contribute().send({
69        value: '200',
```

```
70        from: accounts[1]
71      });
72
73      //Se llama al atributo approvers con la direccion que acaba de realizar la
        contribucion
74      const isContributor = await campaign.methods.approvers(accounts[1]).call()
        ;
75
76      //El valor debe ser true
77      assert(isContributor);
78
79    });
80
81    //Prueba que asegura de requerir una contribucion minima
82    it('Asegura de requerir una contribucion minima', async () => {
83
84      //Se realizar una contribucion menor al monto minimo establecido (100)
85      try {
86        await campaign.methods.contribute().send({
87          value:'5',
88          from: accounts[1]
89        });
90        //Si se llega a este punto de la prueba, es que algo malo ocurrio, la
        idea es que el catch se ejecute
91        assert(false);
92      } catch (err) {
93
94        //Si el catch se ejecuta, hubo un fallo, lo cual es el comportamiento
        esperado
95        assert(err);
96      }
97    });
98
99    //Prueba que asegura que una contribucion minima no puede ser 0
100   it('Asegura que una contribucion minima no puede ser 0', async () => {
101
102     //Se realizar una contribucion de 0
103     try {
104       await campaign.methods.contribute().send({
105         value:'0',
106         from: accounts[1]
107       });
108       //Si se llega a este punto de la prueba, es que algo malo ocurrio, la
        idea es que el catch se ejecute
109       assert(false);
110     } catch (err) {
111
```

```
112       //Si el catch se ejecuta, hubo un fallo, lo cual es el comportamiento
      esperado
113       assert(err);
114     }
115   });
116
117   //Prueba que asegura de no sobrepasar la contribucion maxima
118     it('Asegura de no sobrepasar la contribucion maxima', async () => {
119
120       //Se realizar una contribucion mayor al monto maximo establecido (10
      ether)
121       try {
122         await campaign.methods.contribute().send({
123           value:web3.utils.toWei('15', 'ether'),
124           from: accounts[1]
125         });
126         //Si se llega a este punto de la prueba, es que algo malo ocurrio, la
      idea es que el catch se ejecute
127         assert(false);
128       } catch (err) {
129
130         //Si el catch se ejecuta, hubo un fallo, lo cual es el comportamiento
      esperado
131         assert(err);
132       }
133     });
134
135   //Prueba que permite al gerente crear solicitudes
136   it('Permite al gerente crear solicitudes', async () => {
137
138     //Se llama al metodo 'createRequest'
139     await campaign.methods.createRequest('Comprar baterias', '100', accounts
      [1]).send({
140       from: accounts[0],
141       gas: '1000000'
142     });
143
144     //Se obtiene la solicitud recien creada
145     const request = await campaign.methods.requests(0).call();
146
147     //Se verifica que la descripcion de la solicitud sea 'Comprar baterias'
148     assert.equal('Comprar baterias', request.description);
149   });
150
151   //Prueba que permite procesar solicitudes
152   it('Procesar solicitudes', async () => {
153
154     //Primero se realiza una contribucion
```

```
155     await campaign.methods.contribute().send({
156       from: accounts[0],
157       value: web3.utils.toWei('10', 'ether')
158     });
159
160     //Se crea una solicitud
161     await campaign.methods.createRequest('A', web3.utils.toWei('5', 'ether'),
        accounts[1]).send({
162       from:accounts[0],
163       gas: '1000000'
164     });
165
166     //Se aprueba la solicitud recien creada
167     await campaign.methods.approveRequest(0).send({
168       from:accounts[0],
169       gas: '1000000'
170     });
171
172     //Se finaliza la solicitud recien aprobada
173     await campaign.methods.finalizeRequest(0).send({
174       from:accounts[0],
175       gas:'1000000'
176     });
177
178     //Se obtiene el balance de la cuenta destino
179     let balance = await web3.eth.getBalance(accounts[1]);
180
181     //El balance se pasa de wei a ether
182     balance = web3.utils.fromWei(balance, 'ether');
183     balance = parseFloat(balance);
184
185     console.log(balance);
186
187     //Se verifica que el balance sea mayor a lo que tenia en un principio
188     assert(balance > 104);
189
190   });
191
192   //Prueba que permite procesar solicitudes rechazadas
193   it('Procesar solicitudes rechazadas', async () => {
194
195     //Primero se realiza una contribucion
196     await campaign.methods.contribute().send({
197       from: accounts[0],
198       value: web3.utils.toWei('10', 'ether')
199     });
200
201     //Se crea una solicitud
```

```
202     await campaign.methods.createRequest('A', web3.utils.toWei('5', 'ether'),
        accounts[1]).send({
203       from:accounts[0],
204       gas: '1000000'
205     });
206
207     //Se rechaza la solicitud recien creada
208     await campaign.methods.rejectRequest(0).send({
209       from:accounts[0],
210       gas: '1000000'
211     });
212
213     //Se finaliza la solicitud recien rechazada
214     await campaign.methods.finalizeRequest(0).send({
215       from:accounts[0],
216       gas:'1000000'
217     });
218
219     //Se obtiene el balance de la campana
220     let balance = await web3.eth.getBalance(campaignAddress);
221
222     //El balance se pasa de wei a ether
223     balance = web3.utils.fromWei(balance, 'ether');
224     balance = parseFloat(balance);
225
226     console.log(balance);
227
228     //Se verifica que los fondos contruibuidos sigan en la cuenta
229     assert(balance < 11 && balance > 0);
230
231   });
232
233 });
```

Algoritmo 6.7: Código que realiza pruebas automáticas de despliegue de un contrato y funcionalidades. Versión 0.3

La prueba anterior verifica que una transacción rechazada no envíe los fondos desde el balance de la cuenta de la campaña. Se anexan especificaciones de hardware realizadas para esta prueba y los resultados de la misma:

Componente	Especificación
Sistema Operativo	Mac OS Sierra
Memoria	4GB
Procesador	Mac Mini 2.3 GHz Core i5 (I5-2415M)
Disco Duro	1TB

Tabla 6-7: Especificaciones de hardware para pruebas en Mocha, versión 0.3. Ambiente MacOS

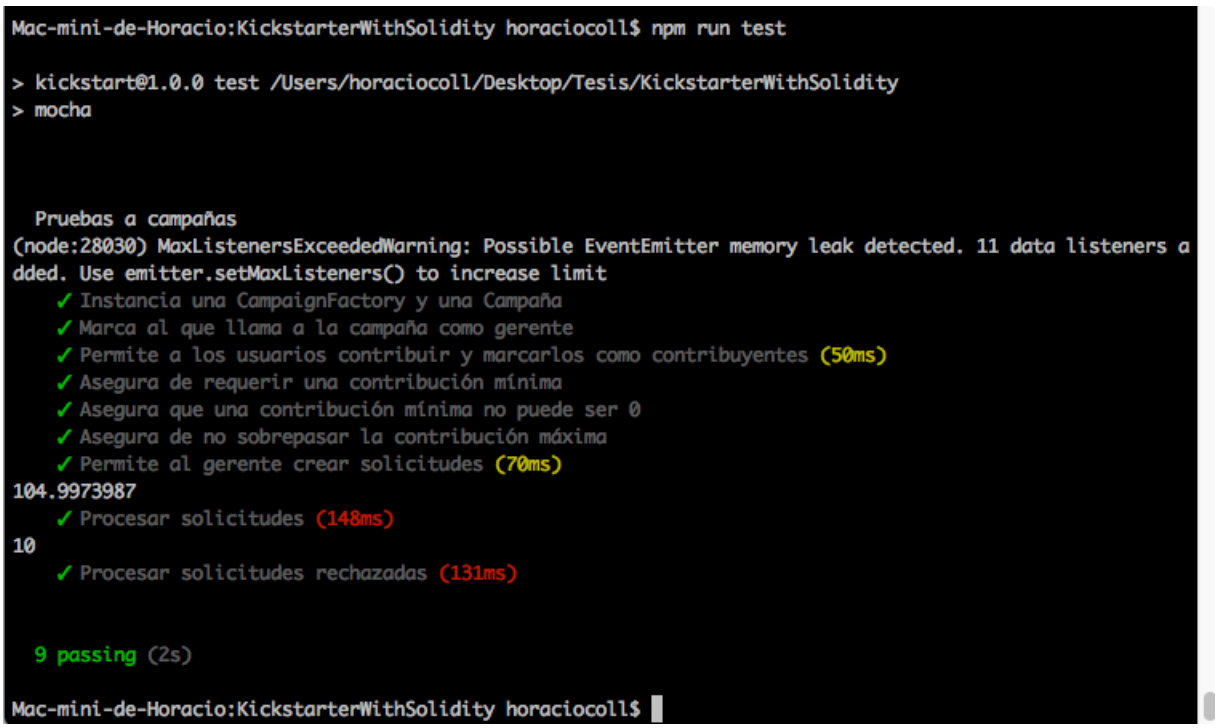

Figura 6-12: Ejecución de pruebas automáticas mediante el uso de la biblioteca Mocha para la versioón 0.3 de la solución..

Luego de verificar que las pruebas son exitosas, la versioón 0.3 del código presenta una estructura bastante avanzada con respecto a la resolución de problemáticas planteadas en el caso de uso de la presente investigación. Sin embargo, queda una lógica que será atacada en la próxima iteración, la expiración de solicitudes.

6.3.5. Iteración 4: Expiración de solicitudes

La iteración 4 afronta el reto de manejar la expiración de solicitudes después de una semana de creadas, la finalidad de este modulo, es que no existan puntos muertos en la recaudación masiva y presionar a los contribuyentes a votar de manera positiva o negativa todas las solicitudes que genere el gerente de la campaña.

A continuación se presenta la versión 1.0 del código, la cual cubre todas las problemáticas planteadas en el caso de uso, por lo cual se considera la primera versión que puede ser desplegada en un ambiente de producción, la estructura final del código en Solidity es:

```
1 //Version de Solidity a utilizar
2 pragma solidity  ^0.4.17;
3
4 //Instancia de un primer contrato, este primer contrato de encarga de generar
    instancias del contrato Campaign
```

```
5  contract CampaignFactory{
6
7      //Ãšnico atributo del contrato generador de campanas, el presente atributo
       se utiliza para devolver todas las direcciones instancias generadas
8      address[] public deployedCampaigns;
9
10     //Funcion que genera instancias del contrato Campaign, recibe un monto
       minimo de contribucion, un monto maximo, un numero maximo de
       contribuyentes, una tasa de aprobacion y una tasa de rechazo
11     function createCampaign(uint minimum, uint maximum, uint maxCont, uint
       approveRate, uint rejectRate) public{
12         address newCampaign = new Campaign(minimum, maximum, maxCont, msg.
       sender, approveRate, rejectRate);
13
14         //Una vez creada la instancia, se agrega al arreglo de direcciones de
       instancias de Campaigns
15         deployedCampaigns.push(newCampaign);
16     }
17
18     //Funcion que retorna un arreglo de direcciones con todas las intancias de
       Campaigns
19     function getDeployedCampaigns()public view returns (address[]){
20         return deployedCampaigns;
21     }
22
23 }
24
25 //Contrato principal, en este contrato se implementa toda la logica y
       funcionalidad de las campanas
26 contract Campaign{
27
28     //Estructura Request que representa una solicitud creada por el gerente
       del contrato
29     struct Request{
30
31         //String que representa la descripcion de la solicitud
32         string description;
33
34         //Entero positivo que representa el valor que desea retirar la
       solicitud de la campana
35         uint value;
36
37         //Direccion que recibira los fondos de la solicitud
38         address recipient;
39
40         //Variable booleana que representa si la solicitud ha sido completada
       o no
41         bool complete;
```

```
42
43          //Entero positivo que representa la cantidad de votos aprobados de la
    solicitud
44          uint approvalCount;
45
46          //Entero positivo que representa la cantidad de votos rechazados de la
     solicitud
47          uint rejectsCount;
48
49          //Entero que representa el momento en que se creo la solicitud
50          uint created;
51
52          //Tipo de dato mapping, que realiza la correspondencia de una
    direccion con un valor booleano, en cuanto se realice una votacion, el
    mapeo de la direccion de esta variable sera asignado a True
53          mapping (address => bool) approvals;
54      }
55
56      //Lista de atributos del contrato
57
58      //Arreglo de solicitudes que posee el contrato
59      Request[] public requests;
60
61      //Direccion del gerente del contrato
62      address public manager;
63
64      //Contribucion minima
65      uint public minimumContribution;
66
67      //Contribucion maxima
68      uint public maximumContribution;
69
70      //Numero maximo de contribuyentes
71      uint public maxContributors;
72
73      //Porcentaje de aprobacion
74      uint public approvalRate;
75
76      //Porcentaje de rechazo
77      uint public rejectedRate;
78
79      //Mapeo de direcciones a valores booleanos que representan la lista de
    contribuyentes votantes
80      mapping (address => bool) public approvers;
81
82      //Contador de votantes
83      uint public approversCount;
84
```

```
85      //Funcion que restringe la posibilidad de una funcion a ser llamada, en
        este caso, restringe a la funcion a ser llamada por alguien que no sea el
        gerente de la campana
86      modifier restricted(){
87          require(msg.sender == manager);
88          _;
89      }
90
91      //Constructor del contrato, recibe sus parametros desde la funcion
        createCampaign de CampaignFactory
92      function Campaign(uint minimum, uint maximum, uint maxCont, address
        creator, uint approveRate, uint rejectRate) public{
93
94          //Es requerido que el monto minimo de aprobacion sea mayor a cero
95          require (minimum > 0);
96
97          //Es requerido que la tasa este entre 0 y 100 (%)
98          require (approveRate > 0 && approveRate < 100);
99
100         //Es requerido que la tasa este entre 0 y 100 (%)
101         require (rejectRate > 0 && rejectRate < 100);
102
103         manager = creator;
104         minimumContribution = minimum;
105         maximumContribution =  maximum;
106         maxContributors = maxCont;
107         approvalRate = approveRate;
108         rejectedRate = rejectRate;
109     }
110
111     //Funcion que permite a una direccion contribuir a la campana, la palabra
        reservada 'payable' indica que la funcion debe recibir un valor en wei, el
         cual sera agregado a la instancia del contrato
112     function contribute() public payable{
113         //Requerimientos basicos para que la funcion pueda seguir su curso
        natural:
114         //-El valor con el que se llama a la funcion debe ser mayor a la
        minima contribucion establecida
115         //-El valor con el que se llama a la funcion debe ser menor a la
        maxima contribucion establecida, o ser cero, para que no haya limite en la
         contribucion
116         //-El nÃºmero de direcciones que aprueban una solicitud no puede ser
        mayor al numero maximo de contribuyentes, o debe ser 0, en caso de que no
        haya limite
117         require (msg.value > minimumContribution &&
118             (msg.value < maximumContribution || maximumContribution == 0 )&&
119             (approversCount < maxContributors || maxContributors == 0));
120
```

```
121         //Este condicional permite a una direccion contribuir varias veces a una misma campana
122         if( approvers[msg.sender] == false){
123 
124           //Una vez que se verifique que la direccion no ha contribuido antes, se mapea su valor a true, y se aumenta el contador
125           approvers[msg.sender] = true;
126           approversCount++;
127         }
128     }
129 
130     //Funcion que crea una solicitud, recibe una descripcion, un valor y una direccion destino (Notese el atributo restricted: solo puede ser llamada por el gerente de la campana)
131     function createRequest(string description, uint value, address recipient) public restricted {
132 
133         //Se crea la nueva estructura con los parametros recibidos
134         Request memory newRequest = Request({
135            description: description,
136            value: value,
137            recipient: recipient,
138 
139            //Se inicializa en falso, puesto que se acaba de crear dicha solicitud y aÃºn no se ha completado
140            complete: false,
141 
142            //Se inicializa el contador de aprobados en cero.
143            approvalCount: 0,
144 
145            //Se inicializa el contador de rechazados en cero.
146            rejectsCount: 0,
147 
148            //Momento de creacion de la solicitud
149            created: now
150         });
151 
152         //Se agrega la solicitud recien creada al arreglo de solicitudes del contrato
153         requests.push(newRequest);
154 
155     }
156 
157     //Funcion llamada por los contribuyentes de la campana, para aprobar una solicitud, recibe un indice del arreglo de solicitudes para conocer cual solicitud desea votar para su aprobacion
158     function approveRequest(uint index) public{
159 
```

```
160         //Localizar la solicitud segun el indice
161         Request storage request = requests[index];
162 
163         //Para que la funcion continue su curso natural, es requerido que la direccion que llama la funcion sea un contribyente
164         require(approvers[msg.sender]);
165 
166         //Tambien es necesario que la direccion que llama a la funcion NO haya votado anteriormente
167         require(!request.approvals[msg.sender]);
168 
169         //La direccion aprobo la solicitud
170         request.approvals[msg.sender] = true;
171 
172         //Se incrementa la cuenta de votos
173         request.approvalCount++;
174 
175     }
176 
177     //Funcion llamada por los contribuyentes de la campana, para rechazar una solicitud, recibe un indice del arreglo de solicitudes para conocer cual solicitud desea votar para su rechazo
178     function rejectRequest(uint index) public{
179 
180         //Localizar la solicitud segun el indice
181         Request storage request = requests[index];
182 
183         //Para que la funcion continue su curso natural, es requerido que la direccion que llama la funcion sea un contribyente
184         require(approvers[msg.sender]);
185 
186         //Tambien es necesario que la direccion que llama a la funcion NO haya votado anteriormente
187         require(!request.approvals[msg.sender]);
188 
189         //La direccion rechazo la solicitud
190         request.approvals[msg.sender] = true;
191 
192         //Se incrementa la cuenta de votos
193         request.rejectsCount++;
194 
195     }
196 
197     //Funcion llamada para dar por finalizada una solicitud, recibe un indice para identificar la solicitud a finalizar, notese el atributo restricted, solo puede ser llamada por el gerente
198     function finalizeRequest(uint index) public restricted {
199         //Identificando la solicitud
```

```
200         Request storage request = requests[index];
201 
202         //Momento en el que se realizo la finalizacion
203         uint finalizedMoment = now;
204 
205         //Variable que refleja la validez de la transaccion
206         bool valid;
207 
208     //Tiempo epoch que representa una semana
209         int week = 604800;
210 
211         //Si la transaccion tiene mas de 1 semana se vuelve invalida
212         int transactionTime = int(request.created) + week - int(
    finalizedMoment);
213 
214         if (transactionTime < 0){
215             valid =  false;
216         }else{
217             valid = true;
218         }
219 
220         //Es requerido que hayan votado positivamente o negativamente la
    cantidad establecida de contribuyentes en el constructor ( Variables
    approvalRate y rejectedRate )
221         require((request.approvalCount > (approversCount*(approvalRate))/100)
    || (request.rejectsCount > (approversCount*(rejectedRate))/100));
222 
223         //Es requerido, ademas, que la solicitud no haya sido marcada como
    completada
224         require(!request.complete);
225 
226         //Si la transaccion es valida, se ejecuta normalmente
227         if (valid){
228             //Si es el caso de aprobacion, se traspasa el ether
229             if(request.approvalCount > (approversCount*(approvalRate))/100){
230 
231                 //Se transfieren los fondos a la direccion destino
232                 request.recipient.transfer(request.value);
233 
234                 //Se marca la solicitud como completada
235                 request.complete = true;
236             }
237 
238             //En caso que haya sido rechazada, se finaliza la solicitud sin
    enviar fondos
239             if(request.rejectsCount > (approversCount*(rejectedRate))/100){
240 
241                 //Se marca la solicitud como completada
```

```
242                 request.complete = true;
243             }
244 
245         //Si no es valida, se envian los fondos a la direccion destino
246         }else{
247 
248             //Se transfieren los fondos a la direccion destino
249             request.recipient.transfer(request.value);
250 
251             //Se marca la solicitud como completada
252             request.complete = true;
253         }
254 
255 
256     }
257 
258     //Funcion que retorna un resumen de la campana
259     function getSummary() public view returns(uint, uint, uint, uint, uint,
    uint, uint, uint, address){
260 
261         //Retornar todos los atributos de la campana
262         return (
263           minimumContribution,
264           maximumContribution,
265           maxContributors,
266           this.balance,
267           requests.length,
268           approversCount,
269           approvalRate,
270           rejectedRate,
271           manager
272           );
273     }
274 
275     //Funcion que retorna el numero de solicitudes
276     function getRequestCount() public view returns(uint){
277         return requests.length;
278     }
279 }
```

Algoritmo 6.8: Versión 1.0 de solución para el caso de uso

La versión 1.0 de la solución involucra toda la lógica con respecto a la expiración de solicitudes.
Lamentablemente las pruebas con respecto a tiempos de expiración se vuelven complicadas debido a su ejecución automática, lo cual no permite la medición de paso de tiempo.

Debido a lo anteriomente expuesto, las pruebas de expiración de solicitudes se realizarán

una vez que el cliente web haya sido desarrollado y se logre la interacción con el contrato mediante el mismo.

6.3.6. Iteración 5: Construcción cliente web

La iteración 5 presenta el desarrollo que permite a un cliente web interactuar con diferentes instancias de un contrato inteligente, este desarrollo presenta una serie de pasos previos que serán explicados en detalle a medida que se vaya avanzando en el despliegue en la red, nótese que se hará uso de los archivos JSON obtenidos a partir del código que realiza la compilación (Algoritmo 6.2)

El cliente web se encargará de generar los contratos en la red de Ethereum Rinkeby, la cual si bien no es la red real de Ethereum, es una red que simula un ambiente de producción de manera que se prueben todas las funcionalidades tal cual como si se estuviera en un ambiente de producción manejando fondos reales, sin el riesgo de perder capital real en la realización de dichas pruebas.

6.3.6.1. Modulo Web3

Una vez obtenido el ABI y *Bytecode* generado, debe generarse y exportarse el siguiente módulo para obtener una instancia de Web3 que interactúe con las cuentas generadas por MetaMask para interactuar con la red Rinkeby:

```
1
2  //Dependencias necesarias
3  import Web3 from 'web3';
4
5  let web3;
6
7  if (typeof window !== 'undefined' && window.web3 !== 'undefined') {
8
9    //Se esta haciendo uso del navegador Y Metamask esta activo
10   web3 = new Web3(window.web3.currentProvider);
11
12 }else {
13
14   //Esta siendo llamado del lado del servidor, o Metamask NO esta activado
15   const provider = new Web3.providers.HttpProvider(
16
17     //Nodo Infura al que se conectara nuestra instancia de Metamask para
       interactuar con la red Rinkeby de Ethereum
18     'https://rinkeby.infura.io/O9rDBBXPQ0LgMzdxly6I'
19   );
20
21   web3 = new Web3(provider);
```

```
}

export default web3;
```

Algoritmo 6.9: Módulo que genera una instancia de Web3 conectada a la red Rinkeby(web3.js)

Este módulo será uno de los componentes principales en todas las interacciones que se realicen con la red Rinkeby.

6.3.6.2. Despliegue del contrato en la red

El siguiente código se encarga se realizar el despliegue de nuestro contrato en la red de Rinkeby, el resultado del mismo, será una dirección de Ethereum, esta dirección representa la instancia de nuestro contrato:

```

//Dependencias necesarias
const HDWalletProvider = require('truffle-hdwallet-provider');
const Web3 = require('web3');

//Se obtiene el ABI y Bytecode
const compiledFactory = require('./build/CampaignFactory.json');

//Proveedor que se conectara a la red Ethereum para instanciar contratos (En este caso: Infura)
const provider = new HDWalletProvider(
  //"Mneumonic" que identifica la cuenta que se enlazara con el nodo Infura para crear la instancia en la red Ethereum
  'sniff permit vehicle mixed help wink amazing dash balance moment antenna useless',
  'https://rinkeby.infura.io/O9rDBBXPQ0LgMzdxly6I'
);

//Instancia de Web3, que recibe 'provider'
const web3 = new Web3(provider);

//Funcion que instanciara el contrato en la red
const deploy  = async () => {
  //Lista de cuentas asociadas al "Mneumonic" en 'provider'
  accounts = await web3.eth.getAccounts();

  //Se muestra por consola, la cuenta que hace despliegue en la red
  console.log('Desplegando desde la direccion: ', accounts[0]);

  //Creacion del contrato en la red.
  const result = await new web3.eth.Contract(JSON.parse(compiledFactory.interface))
```

```
29      .deploy({ data: compiledFactory.bytecode })
30      .send({ gas: '2000000', from : accounts[0]});
31
32    //Se muestra por pantalla la direccion donde el contrato fue creado.
33    console.log('Contrato desplegado en :', result.options.address);
34
35 };
36
37 //Llamada a la funcion
38 deploy();
```

Algoritmo 6.10: Módulo que realiza el despligue del contrato en la red Rinkeby (deploy.js)

Se ejecuta el código mediante la consola, utilizando el comando 'node deploy.js', la ejecución del código generará la dirección del contrato a la que haremos referencia más adelante para interactuar con un contrato, dada su dirección:

```
Mac-mini-de-Horacio:ethereum horaciocoll$ node deploy.js
Desplegando desde la dirección:  0xb5D80826D0c4f8E95Df3f8C5B6D29BA8faEE79aB
Contrato desplegado en : 0xcdc42b2899e526Ff14Fbf5E1cf2A1b055E38A034
Mac-mini-de-Horacio:ethereum horaciocoll$
```

Figura 6-13: Despliegue del contrato en la red Rinkeby, retorna una dirección del contrato desplegado

6.3.6.3. Referencias a contratos desplegados en la red

Una vez obtenida la dirección del contrato que se acaba de desplegar en la red, el siguiente código tiene el objetivo de obtener una instancia con un contrato, dada una dirección:

```
1  //Dependencias necesarias
2  import web3 from './web3';
3  import CampaignFactory from './build/CampaignFactory.json';
4
5  //Obtenemos la instancia del contrato "CampaignFactory" dada la direccion
       obtenida en "deploy.js"
6  const instance = new web3.eth.Contract(
7    JSON.parse(CampaignFactory.interface),
8    '0x85Cbb9066D9d18A48ADc93Dcc09165f75C604C02'
9
10 );
11
12 export default instance;
```

Algoritmo 6.11: Módulo que interactúa con un contrato dada una dirección y su ABI (factory.js)

También se podrán referenciar instancias de contratos mediante pase de parámetros, utilizando el siguiente código:

```
1  //Dependencias necesarias
2  import web3 from './web3';
3  import Campaign from './build/Campaign.json';
4
5  //Codigo utilizado para referirnos a una campana en especifico, dado una
       direccion que es recibida como parametro
6  export default (address) => {
7
8    //Devuelve una instancia de contrato
9    return new web3.eth.Contract(
10     JSON.parse(Campaign.interface),
11     address
12   );
13 };
```

Algoritmo 6.12: Módulo que interactúa con un contrato dada una dirección pasada por parámetro y su ABI (campaign.js)

Una vez creadas todas las herramientas para obtener instancias de contratos, se puede empezar a hacer referencias a dichas instancias mediante un cliente web.

6.3.6.4. Cliente web

La construccón del cliente web se realiza utilizando el *framework* React, tal cual como fue especificado en el Capítulo 4 de la presente investigacón. El mismo presenta una estructura de directorios integrada de la siguiente manera:

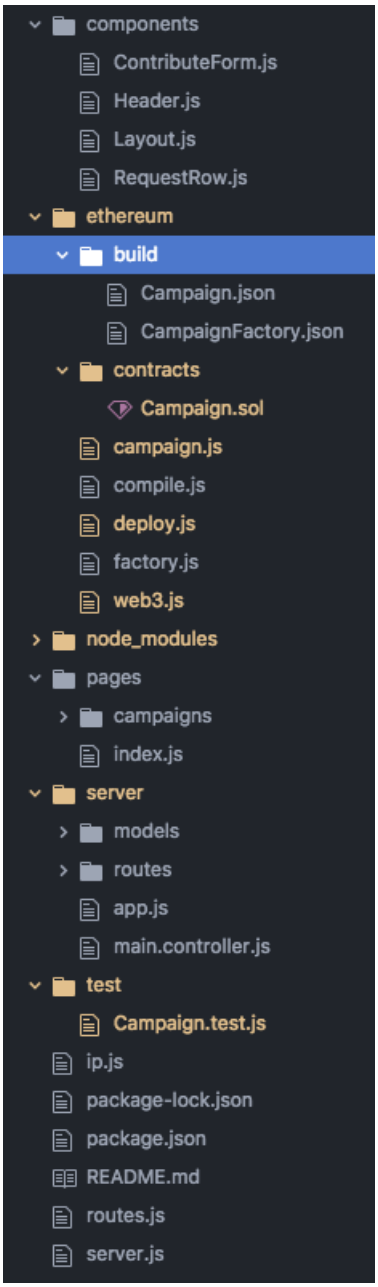

Figura 6-14: Estructura del cliente web

Los diferentes directorios están organizados de la siguiente manera:

- Components: Lista de componentes desarrollados en React para realizar tareas particulares de una manera modularizada.
- Ethereum: Directorio que contiene todos los módulos relaciones con Ethereum y sus interacciones.
- Node modules: Componentes utilizados por Node.
- Pages: Representan las vistas del cliente web, y además representan las rutas de la aplicación gracias al uso de NextJS.
- Server: Dentro de este directorio están definidos los modelos y controladores para el almacenamiento en la base de datos MongoDB.
- Test: Directorio que contiene todas las pruebas realizadas.
- Ip.js: Contiene la ip de la máquina, es utilizado para realizar las solicitudes HTTP para almacenar las interacciones con los contratos en la base de datos.
- Package.json: Lista de bibliotecas y dependencias que utiliza Node.
- Routes.js: Contiene la especificación de las rutas de la aplicación web.
- Server.js: Código a ejecutar para correr en una red local nuestro cliente web.

Una vez descrita la estructura de los directorios de la aplicación web, se contruye el código que despliega la aplicación en una red local, el código tiene la siguiente estructura:

```
1  //Dependencias necesarias
2  const { createServer } = require('http');
3  const next = require('next');
4  const app = next({
5    dev: process.env.NODE_ENV !== 'production '
6  });
7
8  //Lista de modulos necesarios al momento de iniciar la aplicacion web
9
10 //Lado del servidor encargado de manejar el servidor donde se encuentra la
       base datos MongoDB
11 const serverside = require('./server/app.js');
12
13 //Rutas de la aplicacion
14 const routes = require('./routes');
15
16 //Manejo de peticiones
17 const handler = routes.getRequestHandler(app);
```

```
18
19 //Puerto 3000 en el que corre la aplicacion
20 app.prepare().then(() => {
21   createServer(handler).listen(3000, (err) => {
22     if (err) throw err;
23     console.log('Ready on localhost:3000');
24   });
25 });
```

Algoritmo 6.13: Módulo que inicia en una red local la aplicación web(server.js)

Una vez corriendo el servidor, se configura el manejo de rutas mediante el siguiente módulo:

```
1 //Dependencias necesarias
2 const routes = require('next-routes')();
3
4 //Rutas de la aplicacion web
5 routes
6   //:address representa la parte dinamica de la ruta, el segundo parametro,
      representa el archivo en el directorio 'pages' que sera desplegado
7   .add('/campaigns/new', '/campaigns/new')
8   .add('/campaigns/:address', '/campaigns/show')
9   .add('/campaigns/:address/requests', '/campaigns/requests/index')
10  .add('/campaigns/:address/requests/new', '/campaigns/requests/new');
11
12 module.exports = routes;
```

Algoritmo 6.14: Módulo que configura las rutas del servidor web(routes.js)

6.3.6.5. Rutas de la aplicación web

Una vez configuradas las rutas de la aplicación web, NextJS por defecto, asigna el directorio 'Pages' como el directorio de rutas, y el archivo 'index.js' como la ruta inicial cuando se accede a la dirección base (http://localhost:3000). En el archivo 'index.js' podemos observar la lista de campaña desplegadas en la red Rinkeby, además de algunas funcionalidades para navegar dentro de la aplicación, tales como crear campaña o acceder a la información de una campaña. Posee la siguiente estructura:

```
1 //Dependencias de interfaz, rutas y elementos utiles del contrato
2 import React, {Component} from 'react';
3 import { Card, Button } from 'semantic-ui-react';
4 import factory from '../ethereum/factory';
5 import Layout from '../components/Layout';
6 import { Link } from '../routes';
7
8 //Componente principal que renderiza la lista de campanas
9 class CampaignIndex extends Component {
10
11   //Funcion que obtiene los parametros iniciales del componente
```

```
12  static async getInitialProps(){
13
14    //Llamada al contrato para obtener todas las direcciones de los contratos
     existentes
15    const campaigns = await factory.methods.getDeployedCampaigns().call();
16
17    //Se rotorna la lista de contratos, equivale a: return {campaigns:
     campaigns};
18    return { campaigns };
19  }
20
21  //Metodo que itera sobre la lista de campanas y las muestra en la vista
22  renderCampaigns(){
23
24    //Se realiza un map en todas las direcciones de las campanas obtenidas
25    const items = this.props.campaigns.map(address  => {
26      //Se  retorna una direccion y una ruta personalizada con dicha direccion
27      return {
28        header: address,
29        description: (
30          <Link route={`/campaigns/${address}`}>
31            <a> Ver Campana </a>
32          </Link>
33        ),
34        fluid: true
35      }
36    });
37
38    //Card.Group es un elemento de Semantic-UI con vista de tarjetas, recibe
     de parametro el objeto 'items' que contiene todas las campanas
39    return <Card.Group items={items} />;
40  }
41
42  render(){
43    return (
44      <Layout>
45          <div>
46              <h3> Campanas abiertas </h3>
47
48              {/* Boton de crear campana */}
49              <Link route='/campaigns/new'>
50                <a>
51                  <Button
52                      floated = "right"
53                      content= "Crear campana"
54                      icon = "add circle"
55                      primary = {true}
56                    />
```

```
57                  </a>
58                </Link>
59
60                {/* Renderizado de la lista de campanas */}
61                {this.renderCampaigns()}
62              </div>
63          </Layout>
64      )
65    }
66  }
67
68  export default CampaignIndex;
```

Algoritmo 6.15: Ruta que sirve como página base de la aplicación web. Muestra la lista de campañas desplegadas (index.js)

Luego de acceder a la ruta base, se puede navegar hasta la vista de creación de campaña o acceder a los detalles de una campaña en particular.

6.3.6.6. Creación de campañas

El módulo de creación de campañas provee un formulario que permite al usuario (Dicho usuario será el gerente de dicha campaña) ingresar las configuraciones (Monto mínimo de contribución, monto máximo de contribución, número máximo de contribuyentes, tasa de aprobación y tasa de rechazo) con respecto a la campaña que está por crear. El módulo está estructurado de la siguiente manera:

```
1  //Dependencias de interfaz, rutas y elementos utiles del contrato
2  import React, { Component } from 'react';
3  import { Form, Button, Input, Message } from 'semantic-ui-react';
4  import Layout from '../../components/Layout';
5  import factory from '../../ethereum/factory';
6  import web3 from '../../ethereum/web3';
7  import { Router} from '../../routes';
8  import currentIP from '../../ip.js'
9
10 //Componente principal que renderiza el formulario para crear una campana
       nueva
11 class CampaignNew extends Component {
12
13   //Variable 'state' que guardara los datos desde el formulario
14   state = {
15     minimumContribution: '',
16     maximumContribution: '',
17     maximumCont:'',
18     approveRate:'',
19     rejectRate:'',
20     errorMessage: '',
```

```
21      loading: false
22    }
23
24    //Funcion llamada al hacer click en el boton "Crear"
25    onSubmit = async (event) => {
26
27      //Eliminar el comportamiento por defecto de la funcion
28      event.preventDefault();
29
30      //Se activa el atributo 'Loading' del boton mientras que se procesa la
      transaccion en la red
31      this.setState({ loading: true, errorMessage: '' });
32
33      //Inicio de la transaccion
34      try {
35
36        //Se obtiene la cuenta actual de Metamask
37        const accounts = await web3.eth.getAccounts();
38
39        //Se llama al metodo 'createCampaign' del contrato actual y se le pasan
      los parametros obtenidos desde el formulario
40        await factory.methods.createCampaign(this.state.minimumContribution,
41              this.state.maximumContribution,
42              this.state.maximumCont,
43              this.state.approveRate,
44              this.state.rejectRate,
45            ).send({
46            from: accounts[0]
47        });
48
49        //Se obtiene la lista de campanas creadas
50        const lastCampaing =  await factory.methods.getDeployedCampaigns().call
      ();
51
52        //Se realiza la solicitud al servidor donde se encuentra la base de
      datos MongoDB para que guarde la informacion de la campana recien creada
53        fetch('http://' + currentIP  + ':8000/campaign', {
54          method: 'POST',
55          headers: {
56            'Accept': 'application/json',
57            'Content-Type': 'application/json',
58          },
59          body: JSON.stringify({
60            campaignManager: accounts[0],
61            campaignAddress: lastCampaing[lastCampaing.length - 1],
62            minimumContribution: this.state.minimumContribution,
63            maximumContribution: this.state.maximumContribution,
64            maximumCont: this.state.maximumCont,
```

```
65          approveRate:this.state.approveRate,
66          rejectRate:this.state.rejectRate
67        })
68      });
69
70      //Una vez creada la campana, se redirige al usuario a la pagina inicial
    donde podra ver su campana recien creada
71      Router.pushRoute('/');
72
73    } catch (err) {
74
75      //En caso de que ocurra un error, se crear el mensaje de error que se
    mostrara al usuario
76      this.setState({ errorMessage: ['AsegÃºrese de ingresar un nÃºmero
    vÃ¡lido de ether o wei (Sin letras)', 'En caso de ser una lista, no deje
    elementos en blanco', 'Verifique estar usando su plug-in Metamask', '
    Verifique su lista de transacciones pendientes'] });
77    }
78
79    //Finalmente, termina el proceso de 'Loading' del boton
80    this.setState({ loading: false });
81  }
82
83  render(){
84    return(
85      <Layout>
86        <h3> Crear una campana </h3>
87
88        {/* Formulario que rellena el usuario para crear una campana,
    inicialmente no tiene mensaje de error */}
89        <Form onSubmit={this.onSubmit} error={!!this.state.errorMessage}>
90          <Form.Field>
91            <label> Contribucion minima </label>
92
93            {/* Ingresar la contribucion minima */}
94            <Input
95              placeholder='Cantidad minima de wei con la que se puede
    contribuir a la campana'
96              labelPosition='right'
97              label='wei'
98              value={this.state.minimumContribution}
99              onChange={event => this.setState({ minimumContribution: event.
    target.value })}
100           />
101         </Form.Field>
102
103         <Form.Field>
104           <label> Contribucion maxima </label>
```

```
105
106           {/* Ingresar la contribucion maxima */}
107           <Input
108             placeholder='Cantidad maxima de wei con la que se puede
    contribuir a la campana, coloque 0 si no desea establecer un limite maximo
    '
109             labelPosition='right'
110             label='wei'
111             value={this.state.maximumContribution}
112             onChange={event => this.setState({ maximumContribution: event.
    target.value })}
113           />
114         </Form.Field>
115
116         <Form.Field>
117           <label> Numero maximo de contribuyentes </label>
118
119           {/* Ingresar la cantidad maxima de contribuyentes */}
120           <Input
121             placeholder='Cantidad maxima de cotribuyentes que desea en su
    campana, coloque 0 si no desea establecer un limite de contribuyentes'
122             value={this.state.maximumCont}
123             onChange={event => this.setState({ maximumCont: event.target.
    value })}
124           />
125         </Form.Field>
126
127         <Form.Field>
128           <label> Tasa de aprobacion </label>
129
130           {/* Ingresar la tasa de aprobacion*/}
131           <Input
132             placeholder='Porcentaje de votos de aprobacion necesarios para
    aprobar una solicitud, debe ser un numero entre 1 y 99'
133             value={this.state.approveRate}
134             onChange={event => this.setState({ approveRate: event.target.
    value })}
135           />
136         </Form.Field>
137
138         <Form.Field>
139           <label> Tasa de rechazo </label>
140
141           {/* Ingresar la tasa de rechazo*/}
142           <Input
143             placeholder='Porcentaje de votos de rechazo necesarios para
    rechazar una solicitud, debe ser un numero entre 1 y 100'
144             value={this.state.rejectRate}
```

```
145                 onChange={event => this.setState({ rejectRate: event.target.
    value })}
146                 />
147             </Form.Field>
148
149             {/* Mensaje de error que toma la lista de errores desde la funcion '
    onSubmit' */}
150             <Message error header='Hubo un error, tome en cuenta las siguientes
    consideraciones' list={this.state.errorMessage} />
151
152             {/* Boton 'Crear' */}
153             <Button loading={this.state.loading} primary >Â¡Crear!</Button>
154           </Form>
155         </Layout>
156       );
157     }
158
159 }
160
161 export default CampaignNew;
```

Algoritmo 6.16: Ruta que muestra el formulario para la creación de nuevas campañas (campaigns/new.js)

6.3.6.7. Visualizar los detalles de campañas

Cada vez que se cree una nueva campaña, la lista de 'index.js' automáticamente muestra la lista de las campañas creadas, cada campaña, viene acompañada de un botón 'Ver campaña', el cual creará una ruta dinamicamente con la dirección de la campaña seleccionada, el módulo 'show.js', se encarga de mostrar los detalles de una campaña dada una dirección particular. Dicho módulo tiene la siguiente estructura:

```
1  //Dependencias de interfaz, rutas y elementos utiles del contrato
2  import React, { Component } from 'react';
3  import Layout from '../../components/Layout';
4  import Campaign from '../../ethereum/campaign';
5  import { Card, Grid, Button } from 'semantic-ui-react';
6  import web3 from '../../ethereum/web3';
7  import ContributeForm from '../../components/ContributeForm';
8  import { Link } from '../../routes';
9
10 //Componente principal que renderiza las variables de importancia de un
       contrato
11 class CampaignShow extends Component{
12
13   //Funcion que obtiene los parametros iniciales del componente
14   static async getInitialProps(props){
```

```
15
16      //Se obtiene la instancia del contrato dada su direccion, obtenida desde
        la ruta
17      const campaign = Campaign(props.query.address);
18
19      //Se llama al metodo del contrato 'getSummary' que retorna un resumen de
        la campana
20      const summary = await campaign.methods.getSummary().call();
21
22      //Si la contribucion maxima es 0, se reemplaza por '-'
23      if (summary[1] == '0') {
24        summary[1] = '-'
25      }
26
27      //Si el numero maximo de contribuyentes es 0, se reemplaza por '-'
28      if (summary[2] == '0') {
29        summary[2] = '-'
30      }
31
32      //Se obtiene la cuenta activa desde Metamask
33      const accounts = await web3.eth.getAccounts();
34
35      //Se retorna el objeto 'props' que contiene todos los atributos obtenidos
        de la campana
36      return {
37        address: props.query.address,
38        minimumContribution: summary[0],
39        maximumContribution: summary[1],
40        maximumContributors: summary[2],
41        balance: summary[3],
42        requestCount: summary[4],
43        approversCount: summary[5],
44        approvalRate: summary[6],
45        rejectedRate: summary[7],
46        manager: summary[8],
47      };
48    }
49
50    //Metodo que renderiza los campos obtenidos del contrato en tarjetas
        sencillas de leer desde la interfaz
51    renderCards() {
52
53      //Se obtienen los datos desde 'props'
54      const {
55        balance,
56        manager,
57        minimumContribution,
58        maximumContribution,
```

```
59        maximumContributors,
60        requestCount,
61        approversCount,
62        approvalRate,
63        rejectedRate
64      } = this.props;
65
66      //Se crean los items de cada tarjeta, utilizando los objetos obtenidos
      desde 'props'
67      const items = [
68        {
69          header: manager,
70          meta: 'Direccion del gerente',
71          description: 'Gerente que creo la campana, puede crear solicitudes y
      enviar dinero de la campana.',
72          style: {overflowWrap: 'break-word'}
73        },
74        {
75          header: minimumContribution,
76          meta: 'Contribucion minima (wei)',
77          description: 'Para convertirse en contribuyente debe aportar al menos
      esta cantidad de wei.'
78        },
79        {
80          header: maximumContribution,
81          meta: 'Contribucion maxima (wei)',
82          description: 'Para convertirse en contribuyente puede aportar como
      maximo esta cantidad de wei.'
83        },
84        {
85          header: maximumContributors ,
86          meta: 'Cantidad maxima de contribuyentes',
87          description: 'Numero maximo de contribuyentes que pueden aportar a
      esta campana.'
88        },
89        {
90          header: requestCount,
91          meta: 'Numero de solicitudes',
92          description: 'Una solicitud envia fondos de la campana actual. Las
      solicitudes deben ser aprobadas por los contribuyentes.'
93        },
94        {
95          header: approversCount,
96          meta: 'Numero de contribuyentes',
97          description: 'Cantidad de personas que han contribuido a esta campana.
      '
98        },
99        {
```

```
100         //El objeto 'balance' esta en escala Wei, se convierte a Ether con la
    funcion 'FromWei'
101         header: web3.utils.fromWei(balance, 'ether'),
102         meta: 'Balance de la campana (ether)',
103         description: 'El balance de la campana es cuanto dinero tiene esta
    campana para gastar.'
104       },
105       {
106         header: approvalRate + "%",
107         meta: 'Tasa de aprobacion',
108         description: 'Tasa de votos de aprobacion para aprobar una solicitud.'
109       },
110       {
111         header: rejectedRate + "%",
112         meta: 'Tasa de rechazo',
113         description: 'Tasa de votos de rechazo para rechazar una solicitud.'
114       }
115     ];
116
117     //Se retornan las tarjetas con los datos asignados
118     return <Card.Group items={items} />
119   };
120
121   render(){
122     return(
123       <Layout>
124         <h3> Informacion de la campana </h3>
125         <Grid>
126
127           {/* Grid que contiene la lista de tarjetas y el componente de
    contribucion a campanas */}
128           <Grid.Row>
129             <Grid.Column width={10}>
130               {this.renderCards()}
131             </Grid.Column>
132             <Grid.Column width={6}>
133               <ContributeForm address={this.props.address}/>
134             </Grid.Column>
135           </Grid.Row>
136           <Grid.Row>
137             <Grid.Column>
138
139             {/* Boton que lleva a la lista de solicitudes, segun la direccion
    establecida en la ruta */}
140             <Link route={`/campaigns/${this.props.address}/requests`}>
141               <a>
142                 <Button primary>
143                   Ver solicitudes
```

```
144                     </Button>
145                   </a>
146                 </Link>
147                 </Grid.Column>
148               </Grid.Row>
149             </Grid>
150           </Layout>
151       );
152     }
153 }
154
155 export default CampaignShow;
```

Algoritmo 6.17: Ruta que muestra los detalles de una campaña dada una dirección particular (campaigns/show.js)

La vista renderizada por el módulo anteriomente descrito, muestra todos los detalles de importancia de la campaña, adicionalmente, posee un componente que renderiza un botón junto a un campo que un usuario puede llenar y seleccionar para convertirse en un contribuyente. También existe el botón que permite visualizar las solicitudes pendientes de la campaña actual.

6.3.6.8. Contribuir a una campaña

La contribución a una campaña se contruye en un módulo separado del archivo 'show.js' debido a que la contribución tiene su propia lógica especial que debe ser manejada de manera separada a la visualización de los atributos de la campaña.

El módulo de contribución está estructurado de la siguiente manera:

```
1  //Dependencias de interfaz, rutas y elementos utiles del contrato
2  import React, { Component } from 'react';
3  import { Form, Input, Message, Button } from 'semantic-ui-react';
4  import Campaign from '../ethereum/campaign';
5  import web3 from '../ethereum/web3';
6  import { Router } from '../routes';
7  import currentIP from '../ip.js'
8
9  //Componente principal que renderiza el formulario para contribuir a una
       campana
10 class ContributeForm extends Component {
11
12   //Variable 'state' que guardara los datos desde el formulario
13   state = {
14     value:'',
15     errorMessage:'',
16     loading: false
```

```
17  };
18
19  //Funcion llamada al hacer click en el boton 'Contribuir'
20  onSubmit = async (event) => {
21
22    //Eliminar el comportamiento por defecto de la funcion
23    event.preventDefault();
24
25    //Se obtiene una instancia de la campana actual mediante el pase de
      parametros ocurrido en 'show.js'
26    const campaign = Campaign(this.props.address);
27
28    //Se activa el atributo 'Loading' del boton mientras que se procesa la
      transaccion en la red
29    this.setState( {loading: true, errorMessage:'' } );
30
31    //Se obtiene la cuenta actual de Metamask
32    const accounts = await web3.eth.getAccounts();
33
34    try {
35
36      //Se llama al metodo 'contribute' del contrato actual, con los valores
      obtenidos desde el input de 'Contribuir'
37      await campaign.methods.contribute().send({
38        from:accounts[0],
39        value: web3.utils.toWei(this.state.value, 'ether')
40      });
41
42      //Se realiza la solicitud al servidor donde se encuentra la base de
      datos MongoDB para que guarde la informacion de la contribucion realizada
43      fetch('http://' + currentIP + ':8000/contribution', {
44        method: 'POST',
45        headers: {
46          'Accept': 'application/json',
47          'Content-Type': 'application/json',
48        },
49        body: JSON.stringify({
50          campaignAddress: this.props.address,
51          value: web3.utils.toWei(this.state.value, 'ether'),
52          fromAddress: accounts[0]
53        })
54      })
55
56      //Una vez realizada la contribucion, se actualiza la pagina actual para
      que el usuario pueda verla
57      Router.replaceRoute(`/campaigns/${this.props.address}`);
58    } catch (err) {
59
```

```
60      //En caso de que ocurra un error, se crear el mensaje de error que se
    mostrara al usuario
61      this.setState({ errorMessage: ['Asegurese de ingresar un numero valido
    de ether o wei(Sin letras)', 'En caso de ser una lista, no deje elementos
    en blanco', 'Verifique estar usando su plug-in Metamask', 'Verifique su
    lista de transacciones pendientes'] });
62
63    }
64
65    //Finalmente, termina el proceso de 'Loading' del boton
66    this.setState({loading: false, value:'' });
67  }
68
69  render(){
70    return(
71
72
73      <Form onSubmit={this.onSubmit} error={!!this.state.errorMessage}>
74        <Form.Field>
75          <label> Monto de contribucion </label>
76
77          {/* Ingresar el monto de la contribucion */}
78          <Input
79            placeholder='Cantidad de ether con la que desea contribuir'
80            value={this.state.value}
81            onChange= {event => this.setState({ value:event.target.value })}
82            label='ether'
83            labelPosition='right'
84          />
85        </Form.Field>
86
87        {/* Mensaje de error que toma la lista de errores desde la funcion '
    onSubmit' */}
88        <Message error header='Hubo un error, tome en cuenta las siguientes
    consideraciones' list={this.state.errorMessage} />
89
90        {/* Boton 'Contribuir' */}
91        <Button primary loading={this.state.loading}>
92          Â¡Contribuir!
93        </Button>
94      </Form>
95    )
96
97
98  }
99
100 };
101
```

```
102 export default ContributeForm;
```

Algoritmo 6.18: Módulo que maneja las contribuciones a una campaña en específico (Components/ContributionForm.js)

Unas vez cubierta la lógica de contribución a una campaña, el siguiente flujo a cubrir, será el de visualizar solicitudes que estén pendientes en una campaña en específico.

6.3.6.9. Visualizar solicitudes

La tarea de renderizar solicitudes viene en paralelo con la capacidad que tienen los contribuyentes y el gerente de interactuar con dichas solicitudes, es decir, la visualización de las solicitudes van de la mano con la aprobación, rechazo y finalización de las diferentes solicitudes de una campaña. Renderizar e interactuar con las solicitudes es la tarea realizada por el módulo 'index.js' (requests/index.js), su estructura es:

```
1  //Dependencias de interfaz, rutas y elementos utiles del contrato
2  import React, { Component } from 'react';
3  import Layout from '../../../components/Layout';
4  import { Link } from '../../../routes';
5  import { Button, Table } from 'semantic-ui-react';
6  import Campaign from '../../../ethereum/campaign';
7  import RequestRow from '../../../components/RequestRow';
8  import web3 from '../../../ethereum/web3';
9
10 //Componente principal que renderiza la tabla que contiene la lista de
       solicitudes
11 class RequestIndex extends Component {
12
13   //Funcion que obtiene los parametros iniciales del componente
14   static async getInitialProps(props){
15
16     //Se obtiene la direccion de la campana actual
17     const { address } = props.query;
18
19     //Se obtiene la instancia de la campana, dada su direccion
20     const campaign = Campaign(address);
21
22     //Se obtienen la lista de cuentas de Metamask
23     const accounts = await web3.eth.getAccounts();
24
25     //Cuenta actual utilizada
26     const currentAccount = accounts[0];
27
28     //Lista de atributos que se obtienen de la campana
29     const campaignManager = await campaign.methods.manager().call();
30     const requestCount = await campaign.methods.getRequestCount().call();
```

```
31      const approversCount = await campaign.methods.approversCount().call();
32      const approvalRate = await campaign.methods.approvalRate().call();
33      const rejectedRate = await campaign.methods.rejectedRate().call();
34
35      //Balance actual del contrato
36      const currentBalance = await web3.eth.getBalance(address);
37
38      //Conversion a ether del balance del contrato
39      const balanceToEther = await web3.utils.fromWei(currentBalance, 'ether');
40
41      //Se obtiene la lista de solicitudes del contrato
42      const requests = await Promise.all(
43        Array(parseInt(requestCount)).fill().map((element, index) => {
44          return campaign.methods.requests(index).call();
45        })
46      );
47
48      //Se retorta el objeto con todas las variables necesarias para renderizar
49      return { address, requests,requestCount, approversCount, campaignManager,
     currentAccount, approvalRate, rejectedRate, balanceToEther  };
50    }
51
52    //Funcion encargada de renderizar una fila de la tabla en un componente
     separado, junto a la lista de atributos necesarios por fila
53    renderRows() {
54      return this.props.requests.map((request, index) => {
55        return <RequestRow
56          key={index}
57          id={index}
58          request={request}
59          address={this.props.address}
60          approversCount={this.props.approversCount}
61          approvalRate={this.props.approvalRate}
62          rejectedRate={this.props.rejectedRate}
63        />
64      });
65    }
66
67    render(){
68
69      //Se obtienen los atributos necesarios del elemento 'Table' (Provisto por
     Semantic-UI)
70      const { Header, Row, HeaderCell, Body } = Table;
71
72      return(
73        <Layout>
74            <h3> Solicitudes </h3>
75
```

```
76          {/* Balance de la campana actual */}
77          <h3> Balance de la campana: {this.props.balanceToEther} Ether </h3>
78
79          {/* Boton para crear una solicitud */}
80          <Link route={`/campaigns/${this.props.address}/requests/new`}>
81            <a>
82              {/* Este boton se inhabilita si la persona que ve la pagina no
    es el gerente, de esta forma, quien no sea el gerente no puede hacer click
     en este boton */}
83              <Button primary floated='right' style={{ marginBottom:10 }}
    disabled={this.props.campaignManager != this.props.currentAccount}>
84                Crear solicitud
85              </Button>
86            </a>
87          </Link>
88
89          <Table>
90            <Header>
91              <Row>
92
93                {/* Cabeceras de la tabla */}
94                <HeaderCell> ID </HeaderCell>
95                <HeaderCell> Fecha creacion </HeaderCell>
96                <HeaderCell> Expiracion </HeaderCell>
97                <HeaderCell> Descripcion </HeaderCell>
98                <HeaderCell> Monto (ether) </HeaderCell>
99                <HeaderCell> Destino </HeaderCell>
100               <HeaderCell> Cuenta de aprobados </HeaderCell>
101               <HeaderCell> Cuenta de rechazados </HeaderCell>
102               <HeaderCell> Rechazar </HeaderCell>
103               <HeaderCell> Aprobar </HeaderCell>
104               <HeaderCell> Finalizar </HeaderCell>
105
106             </Row>
107           </Header>
108           <Body>
109             {/* Llamado al componente que rellenara las filas de la tabla */
    }
110             {this.renderRows()}
111           </Body>
112         </Table>
113
114         {/* Numero de solicitudes encontradas */}
115         <div> Encontradas {this.props.requestCount} solicitudes. </div>
116
117       </Layout>
118     );
119   }
```

```
120 }
121
122 export default RequestIndex;
```

Algoritmo 6.19: Módulo que maneja las solicitudes a una campaña en específico (campaigns/requests/index.js)

El módulo anterior se encarga de la renderización de la tabla que contiene todas las solicitudes de la campaña, sin embargo, cada una de las filas corresponde a cada una de las solicitudes, lo que requiere aplicar una lógica específica a cada una de las filas de la tabla, por esta razón, para mostrar cada de las filas de la tabla, se creó un componente por separado que se encarga de tomar los datos de cada una de las solicitudes e integrarlas dentro de una fila para ser mostrada en el componente anterior, este componente separado, se encuentra ubicado en 'Components/RequestRow.js':

```
1 //Dependencias de interfaz, rutas y elementos utiles del contrato
2 import React, { Component } from 'react';
3 import { Table, Button, Message } from 'semantic-ui-react';
4 import web3 from '../ethereum/web3';
5 import Campaing from '../ethereum/campaign';
6 import { Router, Link } from '../routes';
7 import currentIP from '../ip.js'
8
9 //Componente principal que renderiza la fila que contiene informacion de cada
    solicitud
10 class RequestRow extends Component {
11
12   //Variable 'state' que guarda los datos de 'loading' y mensaje de erroe
      cuando se haga click en 'Aprobar','Rechazar' o 'Finalizar'
13   state = {
14     loading: false,
15     errorMessage: ''
16   };
17
18   //Funcion que convierte una fecha en Epoch en formato standard
19   epochToDate (epoch) {
20       let date = new Date(epoch*1000);
21       let formattedDate = date.getUTCDate() + '-' + (date.getUTCMonth() + 1)+
     '-' + date.getUTCFullYear();
22       return formattedDate;
23   }
24
25   //Funcion llamada al hacer click en el boton "Aprobar"
26   onApprove = async () => {
27
28     //Se obtiene la instancia del contrato actual dada la direccion desde la
     ruta
29     const campaign =  Campaing(this.props.address);
```

```
30
31    //Se activa el atributo 'Loading' del boton mientras que se procesa la
      transaccion en la red
32    this.setState({ loading: true, errorMessage: '' })
33
34    try {
35
36      //Se obtiene la cuenta actual de Metamask
37      const accounts = await web3.eth.getAccounts();
38
39      //Se llama al mÃ©todo 'approveRequest' del contrato
40      await campaign.methods.approveRequest(this.props.id).send({
41        from: accounts[0]
42      });
43
44      //Se realiza la solicitud al servidor donde se encuentra la base de
      datos MongoDB para que guarde la informacion de la aprobacion realizada
45      fetch('http://' + currentIP  + ':8000/approved', {
46        method: 'POST',
47        headers: {
48          'Accept': 'application/json',
49          'Content-Type': 'application/json',
50        },
51        body: JSON.stringify({
52          campaign: this.props.address,
53          approverAddress: accounts[0],
54          id: this.props.id
55        })
56      })
57
58      //Finalizada la transaccion, se recarga la pagina para que el usuario
      vea su aprobacion o rechazo renderizado
59      Router.replaceRoute(`/campaigns/${this.props.address}/requests`);
60    } catch (err) {
61
62      //En caso de que ocurra un error, se crear el mensaje de error que se
      mostrara al usuario
63      this.setState( {errorMessage: err.message} );
64    }
65
66    //Finalmente, termina el proceso de 'Loading' del boton
67    this.setState({ loading: false, errorMessage: '' })
68  };
69
70  //Funcion llamada al hacer click en el boton "Rechazar"
71  onReject = async () => {
72
73    //Se obtiene la instancia del contrato actual dada la direccion desde la
```

```
    ruta
74    const campaign =  Campaing(this.props.address);
75
76    //Se activa el atributo 'Loading' del boton mientras que se procesa la
    transaccion en la red
77    this.setState({ loading: true, errorMessage: '' })
78
79    try {
80
81      //Se obtiene la cuenta actual de Metamask
82      const accounts = await web3.eth.getAccounts();
83
84      //Se llama al metodo 'rejectRequest' del contrato
85      await campaign.methods.rejectRequest(this.props.id).send({
86        from: accounts[0]
87      });
88
89      //Se realiza la solicitud al servidor donde se encuentra la base de
    datos MongoDB para que guarde la informacion del rechazo realizado
90      fetch('http://' + currentIP  + ':8000/rejected', {
91        method: 'POST',
92        headers: {
93          'Accept': 'application/json',
94          'Content-Type': 'application/json',
95        },
96        body: JSON.stringify({
97          campaign: this.props.address,
98          approverAddress: accounts[0],
99          id: this.props.id
100       })
101     })
102
103     //Finalizada la transaccion, se recarga la pagina para que el usuario
    vea su aprobacion o rechazo renderizado
104     Router.replaceRoute(`/campaigns/${this.props.address}/requests`);
105
106   } catch (err) {
107     //En caso de que ocurra un error, se crear el mensaje de error que se
    mostrara al usuario
108     this.setState( {errorMessage: err.message} );
109   }
110
111   //Finalmente, termina el proceso de 'Loading' del boton
112   this.setState({ loading: false, errorMessage: '' })
113
114
115 };
116
```

```
117  onFinalize = async () => {
118
119    //Se obtiene la instancia del contrato actual dada la direccion desde la
     ruta
120    const campaign =  Campaing(this.props.address);
121
122    //Se activa el atributo 'Loading' del boton mientras que se procesa la
     transaccion en la red
123    this.setState({ loading: true, errorMessage: '' })
124
125    try {
126
127      //Se obtiene la cuenta actual de Metamask
128      const accounts = await web3.eth.getAccounts();
129
130      //Se llama al metodo 'finalizeRequest' del contrato
131      await campaign.methods.finalizeRequest(this.props.id).send({
132        from: accounts[0]
133      });
134
135      //Se realiza la solicitud al servidor donde se encuentra la base de
     datos MongoDB para que guarde la informacion de la finalizacion realizada
136      fetch('http://' + currentIP  + ':8000/finalized', {
137        method: 'POST',
138        headers: {
139          'Accept': 'application/json',
140          'Content-Type': 'application/json',
141        },
142        body: JSON.stringify({
143          campaign: this.props.address,
144          campaignManager: accounts[0],
145          id: this.props.id
146        })
147      })
148
149      //Finalizada la transaccion, se recarga la pagina para que el usuario
     vea su aprobacion o rechazo renderizado
150      Router.replaceRoute(`/campaigns/${this.props.address}/requests`);
151
152    } catch (err) {
153
154      //En caso de que ocurra un error, se crear el mensaje de error que se
     mostrara al usuario
155      this.setState( {errorMessage: err.message});
156    }
157
158    //Finalmente, termina el proceso de 'Loading' del boton
159    this.setState({ loading: false, errorMessage: '' });
```

```
160
161
162  };
163
164  render(){
165
166    //En el metodo render, asignamos varibales de importancia que daran
     informacion al usuario con respecto a cada solicitud
167    const { Row, Cell } = Table;
168    const {id, request, approversCount, approvalRate, rejectedRate} = this.
     props;
169
170    //Validacion sobre si la solicitud esta lista para ser aprobada, se
     mostrara de color verde si este valor es 'true'
171    const readyToApprove = request.approvalCount > approversCount*(
     approvalRate)/100;
172
173    //Validacion sobre si la solicitud esta lista para ser rechazada, se
     mostrara en color naranja si este valor es 'true'
174    const readyToReject = request.rejectsCount > approversCount*(rejectedRate)
     /100;
175
176    //Se obtiene la fecha actual, y se verifica si la solicitud esta expirada,
      en caso afirmativo, la solicitud se mostrara en color rojo
177    const currentTime =  Math.floor((new Date).getTime()/1000);
178    const expired = ((request.created - currentTime + 604800) < 0);
179    console.log(expired);
180
181    return(
182      <Row  positive={readyToApprove && !request.complete} negative={expired}
     warning={readyToReject && !request.complete}>
183        <Cell disabled={request.complete}> {id} </Cell>
184        <Cell disabled={request.complete} collapsing = {true}> {this.
     epochToDate(request.created)} </Cell>
185        <Cell disabled={request.complete} collapsing = {true} negative={
     expired} > {this.epochToDate(parseFloat(request.created) + 604800)} </Cell
     >
186        <Cell disabled={request.complete}> {request.description} </Cell>
187        <Cell disabled={request.complete}> {web3.utils.fromWei(request.value,
     'ether')} </Cell>
188        {/* Link que dirige al usuario a la pÃ¡gina de chequeo de cuentas en
     tiempo real de Ethereum */}
189        <Cell> <Link route={`https://rinkeby.etherscan.io/address/${request.
     recipient}`}>
190                  <a target="_blank">
191                      {request.recipient}
192                  </a>
193                </Link>
```

```
194         </Cell>
195         <Cell disabled={request.complete}> {request.approvalCount}/{
    approversCount} </Cell>
196         <Cell disabled={request.complete}> {request.rejectsCount}/{
    approversCount} </Cell>
197         <Cell disabled={request.complete}>
198           {request.complete ? null: (
199               <Button color='red' basic onClick={this.onReject} loading={this.
    state.loading}>
200                   Rechazar
201               </Button>
202           )
203           }
204         </Cell>
205         <Cell disabled={request.complete}>
206           {request.complete ? null: (
207               <Button color='green' basic onClick={this.onApprove} loading={
    this.state.loading}>
208                   Aprobar
209               </Button>
210           )
211           }
212         </Cell>
213         <Cell disabled={request.complete}>
214         {request.complete ? null: (
215             <Button color='teal' basic onClick={this.onFinalize} loading={this
    .state.loading}>
216               Finalizar
217             </Button>
218           )
219
220         }
221         </Cell>
222
223       </Row>
224     );
225   }
226
227 }
228
229 export default RequestRow
```

Algoritmo 6.20: Módulo que se encarga de construir cada una de las filas de la tabla de solicitudes (Components/RequestRow.js)

6.3.6.10. Creación de solicitudes

El último elemento que compone la solución, viene siendo la creación de solicitudes mediante el cliente web, en la ruta 'requests/index.js' se encuentra el botón de crear solicitud, una vez presionado este botón, el formulario para crear una solicitud será renderizado en la pantalla mediante el siguiente módulo:

```
1  //Dependencias de interfaz, rutas y elementos utiles del contrato
2  import React, { Component } from 'react';
3  import { Form, Button, Message, Input } from 'semantic-ui-react';
4  import Campaign from '../../../ethereum/campaign';
5  import web3 from '../../../ethereum/web3';
6  import { Link, Router } from  '../../../routes';
7  import Layout from '../../../components/Layout';
8  import currentIP from '../../../ip.js'
9
10
11 //Componente principal que renderiza el formulario para crear una solicitud
       nueva
12 class RequestNew extends Component{
13
14   //Variable 'state' que guardara los datos desde el formulario
15   state = {
16       value:'',
17       description:'',
18       recipient:'',
19       loading: false,
20       errorMessage: ''
21   };
22
23   //Funcion que obtiene los parametros iniciales del componente
24   static async getInitialProps(props){
25
26     //Se obtiene la direccion actual del contrato desde la ruta
27     const { address } = props.query;
28     return { address };
29
30   };
31
32   //Funcion llamada al hacer click en el boton "Crear"
33   onSubmit = async event => {
34
35     //Eliminar el comportamiento por defecto de la funcion
36     event.preventDefault();
37
38     //Se obtiene la instancia de la campana con la que se esta trabajando dada
       una direccion
39     const campaign = Campaign(this.props.address);
```

```
40
41      //Se obtienen las variables desde el formulario
42      const { description, value, recipient } = this.state;
43
44      //Se activa el atributo 'Loading' del boton mientras que se procesa la
        transaccion en la red
45      this.setState({ loading: true, errorMessage: '' })
46
47      try {
48
49          //Se obtiene la cuenta actual de Metamask
50          const accounts = await web3.eth.getAccounts();
51
52          //Se llama al metodo 'createRequest' del contrato actual y se le pasan
         los parametros obtenidos desde el formulario
53          await campaign.methods.createRequest(description, web3.utils.toWei(
        value, 'ether'), recipient)
54          .send({ from: accounts[0] });
55
56          //Se realiza la solicitud al servidor donde se encuentra la base de
        datos MongoDB para que guarde la informacion de la solicitud recien creada
57          fetch('http://' + currentIP  + ':8000/request', {
58            method: 'POST',
59            headers: {
60              'Accept': 'application/json',
61              'Content-Type': 'application/json',
62            },
63            body: JSON.stringify({
64              campaignManager: accounts[0],
65              toAddress: recipient,
66              value: web3.utils.toWei(value, 'ether'),
67              description: description,
68              createdAt: Date.now()
69            })
70          });
71
72          //Una creada la solicitud, se redirige al usuario a la lista de
        solicitudes
73          Router.pushRoute(`/campaigns/${this.props.address}/requests`);
74
75      } catch (err) {
76
77        //En caso de que ocurra un error, se crear el mensaje de error que se
        mostrara al usuario
78        this.setState({ errorMessage: ['Asegurese de ingresar un numero valido
        de ether o wei (Sin letras)', 'En caso de ser una lista, no deje elementos
         en blanco', 'Verifique estar usando su plug-in Metamask', 'Verifique su
        lista de transacciones pendientes'] })
```

```
79    }
80
81    //Finalmente, termina el proceso de 'Loading' del boton
82    this.setState({ loading: false })
83
84  };
85
86  render(){
87    return(
88      <Layout>
89
90        {/* Boton para volver a la lista de solicitudes */}
91        <Link route={`/campaigns/${this.props.address}/requests`}>
92          <a>
93              Atras
94          </a>
95        </Link>
96
97        <h3> Crear una nueva solicitud </h3>
98
99        {/* Formulario que rellena el usuario para crear una solicitud,
    inicialmente no tiene mensaje de error */}
100       <Form onSubmit={this.onSubmit} error={!!this.state.errorMessage}>
101         <Form.Field>
102           <label> Descripcion </label>
103
104           {/* Ingresar la descripcion de la solicitud */}
105           <Input
106             placeholder='Descripcion de su solicitud'
107             value={this.state.description}
108             onChange={event => this.setState({ description: event.target.
    value })}
109           />
110         </Form.Field>
111         <Form.Field>
112           <label> Monto en ether</label>
113
114           {/* Ingresar el monto de la solicitud */}
115           <Input
116             labelPosition='right'
117             label='ether'
118             placeholder='Monto que desea retirar de la campana'
119             value={this.state.value}
120             onChange={event => this.setState({ value: event.target.value })}
121            />
122         </Form.Field>
123         <Form.Field>
124           <label> Destino </label>
```

```
125
126             {/* Ingresar la direccion destino la solicitud */}
127             <Input
128               placeholder='Direccion destino a la que iran los fondos de su
    solicitud'
129               value={this.state.recipient}
130               onChange={event => this.setState({ recipient: event.target.value
     })}
131              />
132           </Form.Field>
133
134           {/* Mensaje de error que toma la lista de errores desde la funcion '
    onSubmit' */}
135           <Message error header='Hubo un error, tome en cuenta las siguientes
    consideraciones' list={this.state.errorMessage} />
136
137           {/* Boton 'Crear' */}
138           <Button primary loading={this.state.loading}>
139             Â¡Crear!
140           </Button>
141         </Form>
142       </Layout>
143     );
144   }
145
146 }
147
148 export default RequestNew;
```

Algoritmo 6.21: Módulo que se encarga de construir el formulario para crear una solicitud (pages/campaigns/requests/news.js)

6.3.6.11. Elementos de interfaz

Finalmente, existen dos componentes más que forman parte del cliente web, el primero de ellos, crea una cabecera en el cliente, el cual es reciclado en todas las vistas para darle mejor aspecto físico y congruencia. El mismo está estructurado de la siguiente manera:

```
1  //Dependencias de interfaz, rutas y elementos utiles del contrato
2  import React from 'react';
3  import { Menu } from 'semantic-ui-react';
4  import { Link } from '../routes';
5
6  export default () => {
7    return(
8
9      <Menu style = {{ marginTop: '10px' }} >
10       <Link route='/'>
```

```
11          <a className='item'>
12            Financiacion Colectiva
13          </a>
14        </Link>
15
16        <Menu.Menu position= "right">
17          <Link route='/'>
18            <a className='item'>
19              Campanas
20            </a>
21          </Link>
22
23          <Link route='/campaigns/new'>
24            <a className='item'>
25              +
26            </a>
27          </Link>
28
29
30        </Menu.Menu>
31      </Menu>
32    );
33  }
```

Algoritmo 6.22: Componente que hace de cabecera a todas las páginas del cliente web (Components/Header.js)

El segundo componente se encarga de centrar todos los elementos de la página para dejar margenes a ambos lados para un mejor aspecto y además, es el componente encargado de realizar la llamada al *CDN* de Semantic-UI para su uso:

```
1  //Dependencias de interfaz, rutas y elementos utiles del contrato
2  import React from 'react';
3  import { Container } from 'semantic-ui-react';
4  import Head from 'next/head';
5  import Header from './Header';
6
7  export default (props) => {
8    return (
9
10
11     <Container>
12       <Head>
13         <link rel="stylesheet" href="//cdnjs.cloudflare.com/ajax/libs/semantic
     -ui/2.2.12/semantic.min.css"></link>
14       </Head>
15       <Header />
16         {props.children}
17
```

```
18      </Container>
19    );
20  };
```

Algoritmo 6.23: Componente que hace contenedor a todas las páginas del cliente web (Components/Layout.js)

6.3.7. Iteración 6: Almacenamiento de interacciones

Mientras se desarrollaba el cliente web, se tomó en cuenta el almacenamiento de las interacciones en la base de datos mediante solicitudes HTTP, las interacciones a ser almacenadas son:

- Creación de campañas.
- Contribuciones.
- Solicitudes creadas.
- Aprobaciones.
- Rechazos.
- Finalizaciones.

Una vez definidas estas interacciones, en la presente iteración se desarrollan los módulos que se encargan de construir los modelos que reciben estos datos de parte del cliente y los almacenan en la base de datos, y además se define el módulo controlador de dichas rutas.

6.3.7.1. Servidor MongoDB

A continuación se presenta el módulo que realiza la conexión con una instancia de MongoDB en una red local, además de definir las rutas que controlan las interacciones con esta base de datos:

```
1  //Uso de express
2  const express = require('express');
3  const app = express();
4
5  //Mongo Plugin
6  var mongoose = require("mongoose");
7
8  //Conexion a la base de datos
9  mongoose.connect('mongodb://localhost/campaignDB');
10 var db = mongoose.connection;
11 db.on("error", console.error.bind(console, "Connection error:"));
```

```
db.once("open", function() {
    console.log("Conectado a la base de datos.");
});

// Permitir solicitudes HTTP interactuar con el servidor
app.use(function(req, res, next) {
res.header('Access-Control-Allow-Credentials', true);
res.header('Access-Control-Allow-Origin', req.headers.origin);
res.header('Access-Control-Allow-Methods', 'GET,PUT,POST,DELETE');
res.header('Access-Control-Allow-Headers', 'X-Requested-With, X-HTTP-Method-
    Override, Content-Type, Accept, Authorization');
if ('OPTIONS' == req.method) {
     res.sendStatus(200);
 } else {
     next();
 }
});

//Router
const router = require("./routes/router.js")(app);

//Puerto en el que corre el servidor que maneja la base de datos
app.listen(8000, function () {
  console.log('Escuchando en el puerto 8000');
});
```

Algoritmo 6.24: Componente que se conecta con MongoDB (server/app.js)

Una vez que este módulo se ejecuta, se crea la ruta del módulo controlador que se encargará de manejar las solicitudes que accedan al servidor:

```
module.exports = function (app) {

  //Ruta para el procesamiento de solicitudes
    app.use('/', require('../main.controller.js'));
};
```

Algoritmo 6.25: Ruta que asigna el módulo controlador que maneja las solicitudes (server/routes/router.js)

6.3.7.2. Modelos

A continuación se definen los modelos para el almacenamiento en la base de datos:
Campaña:

```
"use strict";
//Dependencia utilizada para la interaccion con la base de datos
var mongoose = require("mongoose");
//Dependencia utilizada para la creacion de 'Schemas' que dan forma a los
    objetos a ser guardados en la base de datos
```

```
var Schema = mongoose.Schema;

//Modelo de creacion de una campana
class campaignModel{

    //Constructor
    constructor(name, schema){
        this.name = name;
        this.schema = schema;
        this.model = mongoose.model(this.name, this.schema);
    }

    //Creacion y guardado del objeto
    createCampaign (campaignObj) {
            var newCampaign = new this.model({
                campaignManager: campaignObj.campaignManager,
                campaignAddress: campaignObj.campaignAddress,
                minimumContribution: campaignObj.minimumContribution,
                maximumContribution: campaignObj.maximumContribution,
                maximumCont: campaignObj.maximumCont,
                approveRate:campaignObj.approveRate,
                rejectRate:campaignObj.rejectRate
            });

            //Guardar en la base de datos
            newCampaign.save(function(err, data) {
                if (err) console.log(err);
                console.log('Campana almacenada');
            });
        }
}

//Modelo de campana a ser guardada en la base de datos
var Campaign = new campaignModel('Campaign', new Schema({
    campaignManager: String,
    campaignAddress: String,
    minimumContribution: String,
    maximumContribution: String,
    maximumCont: String,
    approveRate: String,
    rejectRate: String
}));

module.exports = Campaign;
```

Algoritmo 6.26: Modelo de una campaña para el almacenamiento en la base de datos (server/models/campaign.model.js)

Contribución:

```
"use strict";
//Dependencia utilizada para la interaccion con la base de datos
var mongoose = require("mongoose");
//Dependencia utilizada para la creacion de 'Schemas' que dan forma a los
    objetos a ser guardados en la base de datos
var Schema = mongoose.Schema;

//Modelo de contribucion a una campana
class contributionModel{

    //Constructor
    constructor(name, schema){
        this.name = name;
        this.schema = schema;
        this.model = mongoose.model(this.name, this.schema);
    }

    //Creacion y guardado del objeto
    createContribution (contributionObj) {
            var newContribution = new this.model({
                campaignAddress: contributionObj.campaignAddress,
                value: contributionObj.value,
                fromAddress: contributionObj.fromAddress
            });
            newContribution.save(function(err, data) {
                if (err) console.log(err);
                console.log('Contribucion almacenada');
            });
        }
}

//Modelo de contribucion a ser guardada en la base de datos
var Contribution = new contributionModel('Contribution', new Schema({
    campaignAddress: String,
    value: String,
    fromAddress: String
}));

module.exports = Contribution;
```

Algoritmo 6.27: Modelo de una contribución para el almacenamiento en la base de datos (server/models/contribution.model.js)

Solicitudes:

```
"use strict";
//Dependencia utilizada para la interaccion con la base de datos
var mongoose = require("mongoose");
//Dependencia utilizada para la creacion de 'Schemas' que dan forma a los
    objetos a ser guardados en la base de datos
```

```
var Schema = mongoose.Schema;

//Modelo de solicitud a una campana
class requestModel{

    //Constructor
    constructor(name, schema){
        this.name = name;
        this.schema = schema;
        this.model = mongoose.model(this.name, this.schema);
    }

    //Creacion y guardado del objeto
    createRequest (requestObj) {
            var newRequest = new this.model({
                campaignManager: requestObj.campaignManager,
                toAddress: requestObj.toAddress,
                value: requestObj.value,
                description: requestObj.description,
                createdAt: requestObj.createdAt

            });

            //Guardar en la base de datos
            newRequest.save(function(err, data) {
                if (err) console.log(err);
                console.log('Solicitud almacenada');
            });
        }
}

//Modelo de solicitud a ser guardada en la base de datos
var Request = new requestModel('Request', new Schema({
    campaignManager: String,
    toAddress: String,
    value: String,
    description: String,
    createdAt: Number
}));

module.exports = Request;
```

Algoritmo 6.28: Modelo de una solicitud para el almacenamiento en la base de datos (server/models/request.model.js)

Aprobación de una solicitud:

```
"use strict";
//Dependencia utilizada para la interaccion con la base de datos
var mongoose = require("mongoose");
```

```
//Dependencia utilizada para la creacion de 'Schemas' que dan forma a los objetos a ser guardados en la base de datos
var Schema = mongoose.Schema;

//Modelo de aprobacion de una solicitud
class approvedModel{

    //Constructor
    constructor(name, schema){
        this.name = name;
        this.schema = schema;
        this.model = mongoose.model(this.name, this.schema);
    }


    //Creacion y guardado del objeto
    createApproved (approvedObj) {

            var newApproved = new this.model({
                campaign: approvedObj.campaign,
                approverAddress: approvedObj.approverAddress,
                id: approvedObj.id

            });

            //Guardar en la base de datos
            newApproved.save(function(err, data) {
                if (err) console.log(err);
                console.log('Aprobacion almacenada');
            });
        }
}

//Modelo de aprobacion de solicitud a ser guardada en la base de datos
var Approved = new approvedModel('Approved', new Schema({
    campaign: String,
    approverAddress: String,
    id: String
}));

module.exports = Approved;
```

Algoritmo 6.29: Modelo de una aprobación de una solicitud para el almacenamiento en la base de datos (server/models/approved.model.js)

Rechazo de una solicitud:

```
"use strict";
//Dependencia utilizada para la interaccion con la base de datos
var mongoose = require("mongoose");
```

```
//Dependencia utilizada para la creacion de 'Schemas' que dan forma a los
    objetos a ser guardados en la base de datos
var Schema = mongoose.Schema;

//Modelo de rechazo a solicitud
class rejectedModel{

    //Constructor
    constructor(name, schema){
        this.name = name;
        this.schema = schema;
        this.model = mongoose.model(this.name, this.schema);
    }


    //Creacion y guardado del objeto
    createRejected (rejectedObj) {

            var newRejected = new this.model({
                campaign: rejectedObj.campaign,
                rejecterAddress: rejectedObj.rejecterAddress,
                id: rejectedObj.id

            });

            //Guardar en la base de datos
            newRejected.save(function(err, data) {
                if (err) console.log(err);
                console.log('Rechazo almacenado');
            });
        }
}

//Modelo de solicitud rechazada a ser guardada en la base de datos
var Rejected = new rejectedModel('Rejected', new Schema({
    campaign: String,
    approverAddress: String,
    id: String
}));

module.exports = Rejected;
```

Algoritmo 6.30: Modelo de un rechazo de una solicitud para el almacenamiento en la base de datos (server/models/rejected.model.js)

Finalización de una solicitud:

```
"use strict";
//Dependencia utilizada para la interaccion con la base de datos
var mongoose = require("mongoose");
```

```
//Dependencia utilizada para la creacion de 'Schemas' que dan forma a los objetos a ser guardados en la base de datos
var Schema = mongoose.Schema;

//Modelo de finalizacion de una solicitud
class finalizedModel{

    //Constructor
    constructor(name, schema){
        this.name = name;
        this.schema = schema;
        this.model = mongoose.model(this.name, this.schema);
    }

    //Creacion y guardado del objeto
    createFinalized (finalizedObj) {
            var newFinalized = new this.model({
                campaign: finalizedObj.campaign,
                campaignManager: finalizedObj.campaignManager,
                id: finalizedObj.id

            });

            //Guardar en la base de datos
            newFinalized.save(function(err, data) {
                if (err) console.log(err);
                console.log('Finalizacion almacenada');
            });
        }
}

//Modelo de finalizacion de una solicitud a ser guardada en la base de datos
var Finalized = new finalizedModel('Finalized', new Schema({
    campaign: String,
    campaignManager: String,
    id: String
}));

module.exports = Finalized;
```

Algoritmo 6.31: Modelo de finalización de una solicitud para el almacenamiento en la base de datos (server/models/finalized.model.js)

Finalmente, se define el módulo controlador que se encarga de utilizar todos los modelos previamente definidos para almacenar las diferentes interacciones dependiendo de la dirección a la que está dirigida la solicitud HTPP desde el lado del cliente:

```
//Dependencias necesarias
const express = require('express');
```

```
3  const router = express.Router();
4
5  //Modelos de objetos a guardar en la base de datos
6  const mongoose = require('mongoose');
7  const Contribution = require('./models/contribution.model.js');
8  const Campaign = require('./models/campaign.model.js');
9  const Request = require('./models/request.model.js');
10 const Finalized = require('./models/finalized.model.js');
11 const Approved = require('./models/approved.model.js');
12 const Rejected = require('./models/rejected.model.js');
13
14 //Body Parser para procesar solicitudes HTTP
15 const bodyParser = require("body-parser");
16 router.use(bodyParser.urlencoded({ extended: false }));
17 router.use(bodyParser.json());
18
19 //Ruta que maneja 'contribution', que almacena una contribucion
20 router.post('/contribution', function (req, res) {
21
22   let contributionObj = {};
23   contributionObj.campaignAddress = req.body.campaignAddress;
24   contributionObj.value = req.body.value;
25   contributionObj.fromAddress = req.body.fromAddress;
26   Contribution.createContribution(contributionObj);
27
28 });
29
30 //Ruta que maneja 'campaign', que almacena una campana nueva
31 router.post('/campaign', function (req, res) {
32
33   let campaignObj = {};
34   campaignObj.campaignManager = req.body.campaignManager;
35   campaignObj.campaignAddress = req.body.campaignAddress;
36   campaignObj.minimumContribution = req.body.minimumContribution;
37   campaignObj.maximumContribution = req.body.maximumContribution;
38   campaignObj.maximumCont = req.body.maximumCont;
39   campaignObj.approveRate = req.body.approveRate;
40   campaignObj.rejectRate = req.body.rejectRate;
41
42   Campaign.createCampaign(campaignObj);
43
44 });
45
46 //Ruta que maneja 'request', que almacena una solicitud nueva
47 router.post('/request', function (req, res) {
48
49   let requestObj = {};
50   requestObj.campaignManager = req.body.campaignManager;
```

```
51   requestObj.toAddress = req.body.toAddress;
52   requestObj.value = req.body.value;
53   requestObj.description = req.body.description;
54   requestObj.createdAt = req.body.createdAt;
55
56   Request.createRequest(requestObj);
57
58 });
59
60 //Ruta que maneja 'finalized', que almacena una finalizacion de una solicitud
61 router.post('/finalized', function (req, res) {
62
63   let finalizedObj = {};
64   finalizedObj.campaign = req.body.campaign;
65   finalizedObj.campaignManager = req.body.campaignManager;
66   finalizedObj.id = req.body.id;
67
68   Finalized.createFinalized(finalizedObj);
69
70 });
71
72 //Ruta que maneja 'approved', que almacena una aprobacion de una solicitud
73 router.post('/approved', function (req, res) {
74
75   let approvedObj = {};
76   approvedObj.campaign = req.body.campaign;
77   approvedObj.approverAddress = req.body.approverAddress;
78   approvedObj.id = req.body.id;
79
80   Approved.createApproved(approvedObj);
81
82 });
83
84 //Ruta que maneja 'rejected', que almacena un rechazo de una solicitud
85 router.post('/rejected', function (req, res) {
86
87   let rejectedObj = {};
88
89   rejectedObj.campaign = req.body.campaign;
90   rejectedObj.approverAddress = req.body.approverAddress;
91   rejectedObj.id = req.body.id;
92
93   Rejected.createRejected(rejectedObj);
94
95 });
96
```

```
97 module.exports = router;
```

Algoritmo 6.32: Módulo encargado de manejar las solicitudes que provengan del lado del cliente y almacenar en la base de datos utilizando los modelos definidos (server/main.controller.js)

Una vez implementadas todas las interacciones con la base de datos, se da por terminada la fase de Construcción, ya que fueron resueltas todas las problemáticas planteadas en el caso de uso y se da cabida a la última fase de la metodología en la cual se realizan pruebas de interfaz en ambiente de producción.

6.4. Transición(*Transition*)

Esta última fase de la metodología propone realizar la validación de la solución construida en la fase de construcción. A lo largo de esta fase, se mostrarán las interfaces construidas con el fin de mostrar su usabilidad y validar la propuesta que se mostrará a los usuarios finales (Gerentes y posibles contribuyentes).

6.4.1. Iteración 0: Pruebas de interfaz de usuario

La presente iteración muestra las interfaces de usuario implementadas con el fin de mostrar la usabilidad y funcionalidades, además de describir el comportamiento de la aplicación en un ambiente de producción.
Para iniciar la aplicación web, se ejecuta el comando 'npm run dev' desde la consola, situándose en el directorio de la herramienta.

En vista de comprobar el correcto funcionamiento de las interfaces, se hizo un sondeo a 15 personas con un perfil de usuario final para que pudieran evaluar las interfaces creadas con el fin de comprobar la correcta transmisión de la información. Se muestra una tabla compartida con los usuarios que interactuaron con la aplicación para que pudieran evaluar las interfaces con una calificación del 1 al 5, donde la puntuación representa:

1. Muy mal.
2. Mal
3. Regular.
4. Bien.
5. Excelente.

En la siguiente tabla puede observarse la descripción del tipo de prueba a realizar:

Tipo de prueba	Aceptación
Ejecutantes	Usuarios finales potenciales
Herramienta usada	Preguntas evaluativas
Veces ejecutada	15 por interfaz seleccionada
Resultados Esperados	90 % de resultados mayores a la media evaluativa (Mayores a 2.5)

Tabla 6-8: Especificaciones de la prueba de aceptacón para interfaces gráficas.

Pregunta	Puntuación
¿Considera que la interfaz es agradable a la vista?	
¿La información se entiende correctamente?	
¿Comprende el objetivo de la interfaz?	
¿La funcionalidad de los botones queda clara?	
¿Se siente ubicado en la pantalla actual de la interfaz?	

Tabla 6-9: Tabla para evaluación de interfaces gráficas.

Al momento de evaluar una interfaz, se muestran las preguntas realizadas a los usuarios, luego se encuentra el promedio de las respuestas recibidas por los mismos.

A continuación se muestra la pantalla principal de la herramienta:

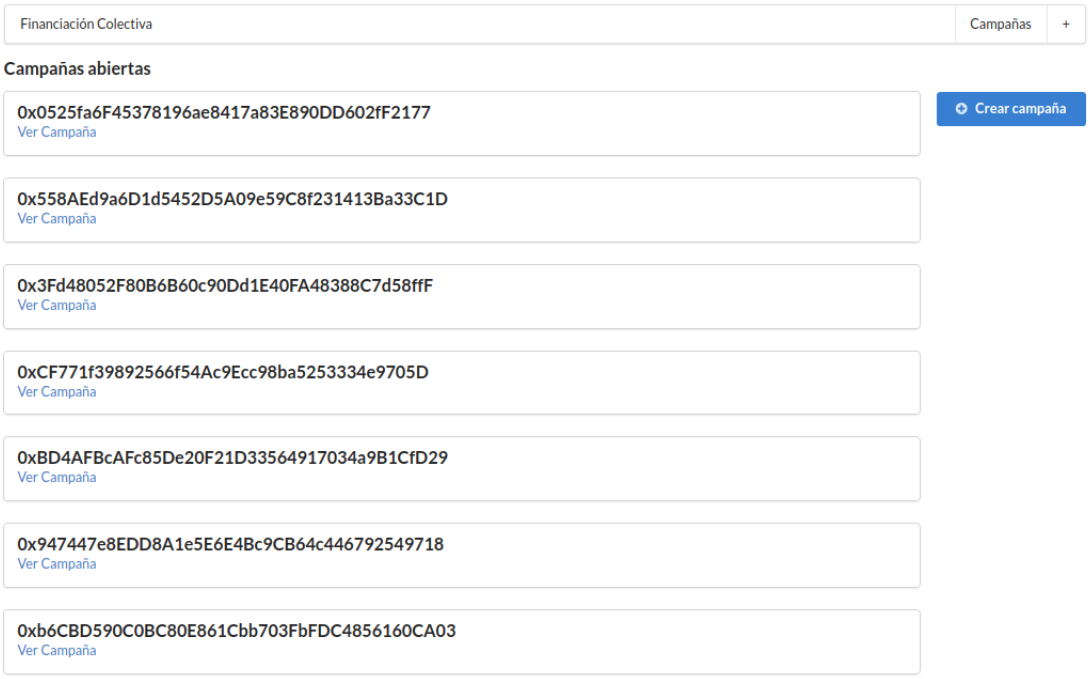

Figura 6-15: Pantalla principal del cliente web.

Pregunta	Puntuación
¿Considera que la interfaz es agradable a la vista?	4.9
¿La información se entiende correctamente?	4.5
¿Comprende el objetivo de la interfaz?	5
¿La funcionalidad de los botones queda clara?	5
¿Se siente ubicado en la pantalla actual de la interfaz?	4.8

Tabla 6-10: Tabla para evaluación de la pantalla principal del cliente web.

6.4.1.1. Crear campaña

La creación de campañas se muestra como un formulario que llena el gerente de la campaña, finalmente se presenta el botón para crear la campaña, una vez creada se mostrará en la pantalla principal de la aplicación:

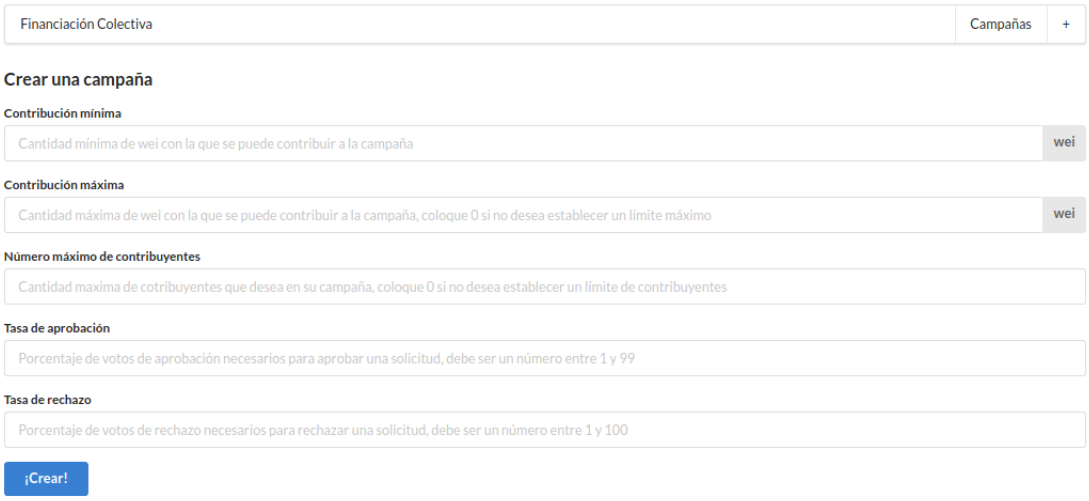

Figura 6-16: Pantalla para la creación de una campaña.

Pregunta	Puntuación
¿Considera que la interfaz es agradable a la vista?	5
¿La información se entiende correctamente?	4.1
¿Comprende el objetivo de la interfaz?	5
¿La funcionalidad de los botones queda clara?	5
¿Se siente ubicado en la pantalla actual de la interfaz?	5

Tabla 6-11: Tabla para evaluación de la pantalla creación de una campaña.

6.4.1.2. Visualizar una campaña / Contribuir a una campaña

Las funcionalidades de visualizar una campaña y contribuir a una campaña vienen una misma vista, debido a que un posible contribuyente querrá tener a su alcance todos los detalles de la campaña a la que contribuirá:

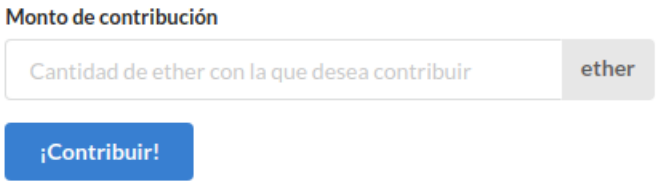

Figura 6-17: Botón de contribución para una campaña.

Pregunta	Puntuación
¿Considera que la interfaz es agradable a la vista?	5
¿La información se entiende correctamente?	4.1
¿Comprende el objetivo de la interfaz?	4.2
¿La funcionalidad de los botones queda clara?	4.9
¿Se siente ubicado en la pantalla actual de la interfaz?	4.0

Tabla 6-12: Tabla para evaluación de botón de contribución para una campaña

Figura 6-18: Detalles de una campaña.

Pregunta	Puntuación
¿Considera que la interfaz es agradable a la vista?	5
¿La información se entiende correctamente?	4.9
¿Comprende el objetivo de la interfaz?	4.5
¿La funcionalidad de los botones queda clara?	4.7
¿Se siente ubicado en la pantalla actual de la interfaz?	4.6

Tabla 6-13: Tabla para evaluación de detalles de una campaña.

6.4.1.3. Crear una solicitud

La vista de creación de solicitudes es muy similar a la de creación de campañas un formulario simple, con un botón de creación de solicitud:

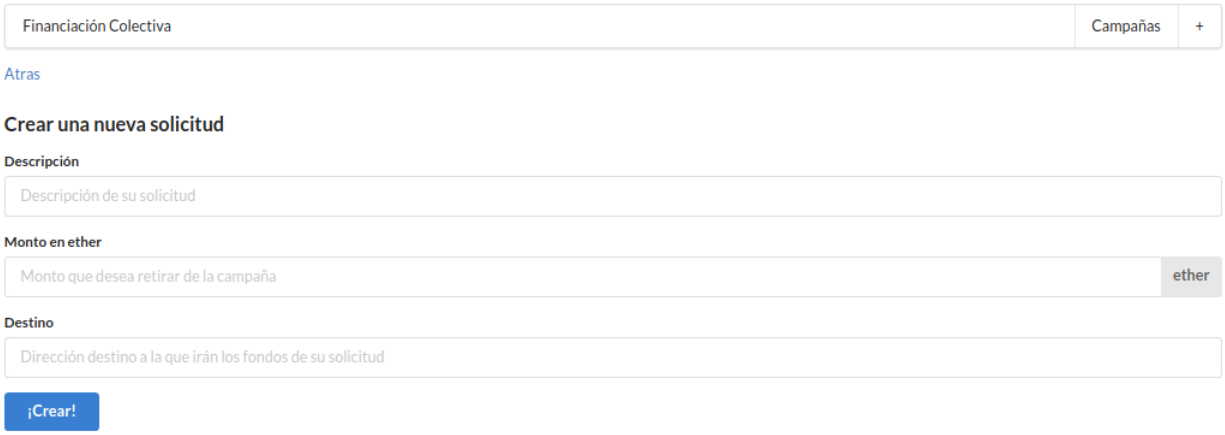

Figura 6-19: Creación de una solicitud.

Pregunta	Puntuación
¿Considera que la interfaz es agradable a la vista?	5
¿La información se entiende correctamente?	4.8
¿Comprende el objetivo de la interfaz?	4.9
¿La funcionalidad de los botones queda clara?	5
¿Se siente ubicado en la pantalla actual de la interfaz?	4.2

Tabla 6-14: Tabla para evaluación de creación de una solicitud.

6.4.1.4. Aprobar/Rechazar/Finalizar solicitudes

Las solicitudes son mostradas mediante una tabla, las tablas tendrán diferentes colores dependiendo del estado de la solicitud:

- Verde: Solicitud lista para ser aprobada.
- Naranja: Solicitud lista para ser expirada.
- Rojo: Solicitud expirada.
- Gris: Solicitud completada.

Balance de la campaña: 0.0004 Ether

Crear solicitud

ID	Fecha creación	Expiración	Descripción	Monto (ether)	Destino	Cuenta de aprobados	Cuenta de rechazados	Rechazar	Aprobar	Finalizar
0	12-4-2018	19-4-2018	Pago de baterias, la dirección destino corresponde a la direccion oficial de la Duncan	0.0002	0xb5D80826D0c4f8E95Df3f8C5B6D29BA8faEE79aB	3/4	0/4	Rechazar	Aprobar	Finalizar
1	12-4-2018	19-4-2018	Pago de facturas a proveedores	0.0001	0xb5D80826D0c4f8E95Df3f8C5B6D29BA8faEE79aB	0/4	3/4	Rechazar	Aprobar	Finalizar
2	12-4-2018	19-4-2018	Compra de aluminio	0.0001	0x1e5d4689fBdceE9695744e6eE58bD673D13d09E4	2/4	2/4	Rechazar	Aprobar	Finalizar
3	12-4-2018	19-4-2018	Compra de plástico a PVC	0.0001	0x8205df7b49c9926F923F6bDfB180e7E2e40c757F	0/4	0/4	Rechazar	Aprobar	Finalizar

Figura 6-20: Lista de solicitudes, una lista para aprobar, una lista para ser rechazada, dos en vigencia.

Pregunta	Puntuación
¿Considera que la interfaz es agradable a la vista?	4.9
¿La información se entiende correctamente?	4.2
¿Comprende el objetivo de la interfaz?	5
¿La funcionalidad de los botones queda clara?	5
¿Se siente ubicado en la pantalla actual de la interfaz?	4.7

Tabla 6-15: Tabla para evaluación de lista de solicitudes.

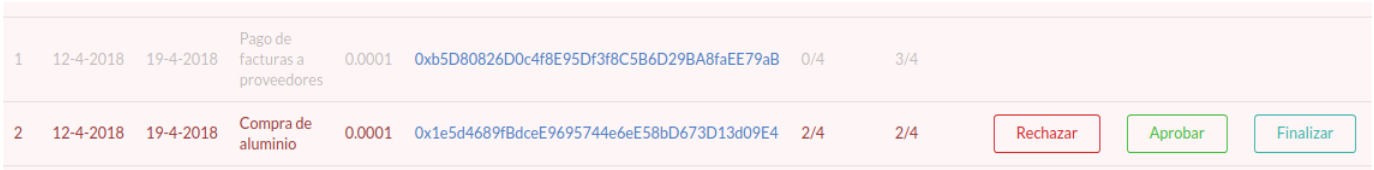

Figura 6-21: Lista de solicitudes, una expirada, una finalizada.

En vista de que las pruebas arrojaron valores por encima de los valores definidos en el tipo de prueba se asume que la construcción de las interfaces se realizó de una manera coherente y la información es transmitida de una manera eficaz.

6.4.1.5. Validación y mensajes de error

A continuación se listan los mensajes de error y validación en caso de que algún usuario cometa un error al momento de ingresar datos en la aplicación web:

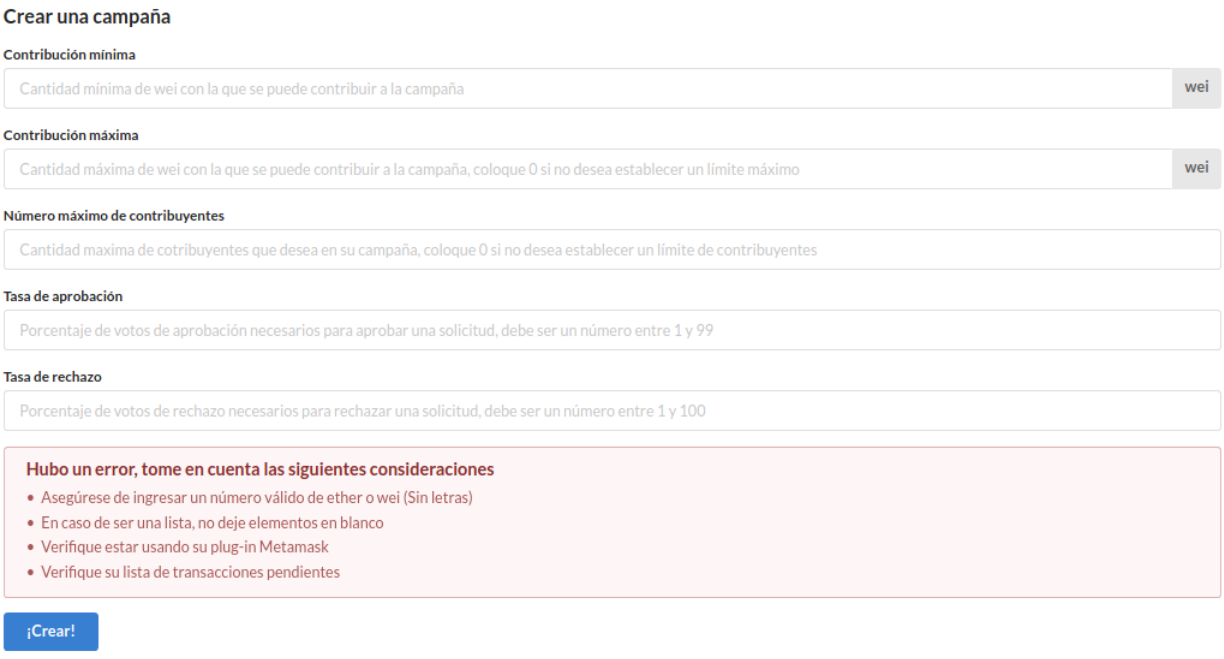

Figura 6-22: Error al dejar campos vacíos en la creación de una campaña.

El mismo mensaje de error es desplegado al querer ingresar datos incorrectos al momento de ingresar una contribución y al querer ingresar datos inválidos en la creación de una solicitud.

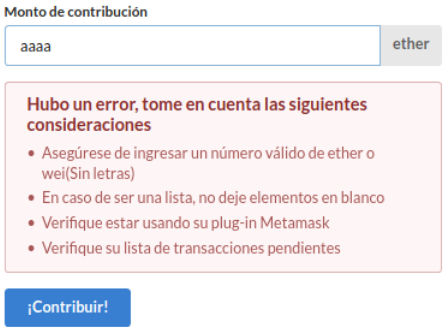

Figura 6-23: Error al insertar caracteres al monto de contribución.

7 Capítulo 7 - Conclusiones

El presente TEG muestra las bondades que brindan estos nuevos enfoques para resolver problemáticas complejas que cada vez requieren más confianza y transparencia a la hora de manejar bienes valiosos como criptomonedas y cualquier tipo de propiedad que represente un valor para el que la posea.

Si bien la solución planteada no es infalible, es un paso más cerca a eliminar las malversaciones de bienes y corrupción en sistemas abiertos que ayudan a financiar ideas.

Se logró crear una solución que, apoyada en la criptografía, las cadenas de bloques y los contratos inteligentes, puede brindar confianza a sus usuarios, y además fomenta la participación colectiva y reincidente de los mismos.

Este tipo de soluciones, son sólo el inicio de una tecnología muy jóven que cada día se refina un poco más y promete revolucionar como se manejan la propiedad del individuo y la confianza que este le tiene a los sistemas distribuidos.

7.0.1. Seleccionar Caso de Uso

Se logró seleccionar un caso de uso que ilustraba de una manera evidente una problemática a resolver, y además, muestra como la criptografía, apoyada en la cadena de bloques y los contratos inteligentes, puede ser una solución eficiente a dichas problemáticas con la finalidad de generar confianza y transparencia a la hora de traspasar fondos.

7.0.2. Seleccionar una metodología de desarrollo

Se planteó una metodología que logró adaptarse y ser eficiente en los enfoques ágiles actuales sin dejar de lado prácticas tradicionales. Finalmente la metodología resultó un pilar fundamental a la hora de definir las etapas en las que se construía la herramienta y la forma en que fueron ejecutadas.

7.0.3. Preparar el ambiente de desarrollo de la plataforma Ethereum

Se hizo uso de las bibliotecas Web3, Ganache, SolC y NodeJS como piezas fundamentales a la hora de preparar ambientes de desarrollo para el uso de contratos inteligentes, ya sea de

manera local o en ambientes de producción. Estas herramientas permitieron la interacción con la red y los contratos inteligentes que en ella residen de una manera rápida y sencilla, se puede concluir que la preparación del ambiente para la herramienta desarrollada fue exitoso gracias al uso de estas herramientas.

7.0.4. Recopilación de datos

La recopilación de las interacciones del usuario con el contrato fueron almacenadas de manera exitosa en la herramienta propuesta, MongoDB probó ser una herramienta versátil para realizar este tipo de tareas. Su similaridad con el uso de objetos de JavaScript hizo que su implementación se lograra de manera sencilla y eficiente.

7.0.5. Definir almacén de datos

El alcance de la investigación no pudo cubrir una definición que pueda considerarse eficiente del punto de vista de un analista de datos. Se logró recopilar ciertos datos de importancia, pero no se pudo asegurar que la forma en que fueron almacenados fue la más eficiente o que fue utilizada la herramienta que mejor se ajuste a este tipo de análisis futuros. El uso, transformación y carga de los datos en plataformas eficientes para el procesamiento de grandes volúmenes de datos se releva a futuras investigaciones.

7.0.6. Implementar la aplicación sobre la base tecnológica de cadena de bloques y contratos inteligentes

La combinación de las tecnologías utilizadas, junto con la metodología seleccionada dieron pie a varias iteraciones que permitieron implementar en fases concretas una solución al caso de uso planteado, la solución permite una interacción directa tanto como con la red Ethereum como con los contratos inteligentes que viven en la misma.

7.0.7. Pruebas de funcionamiento

Se realizaron pruebas de tipo funcional las cuales validaron el correcto funcionamiento de la aplicación en cada una de las versiones, se realizó una prueba por cada versión desplegada y las mismas ayudaron a detectar errores en cada una de las diferentes iteraciones.

Adicionalmente se realizaron pruebas de aceptación las cuales fueron realizadas por potenciales usuarios finales con la finalidad de validar la correcta transferencia de información desde la aplicación a los usuarios finales, los resultados de estas pruebas de validación arrojaron en su totalidad resultados positivos lo cual reafirmó su correcto funcionamiento.

7.0.8. Pruesta en marcha

Finalmente, la puesta en marcha en un ambiente de producción se logró exitosamente mediante el uso de un nodo Infura el cual permitió que la herramienta previamente desarrollada y probada pudiera ser desplegada para que los usuarios finales pudieran interactuar con la herramienta.

7.1. Contribuciones

- Se sentaron las bases para el desarrollo de aplicaciones que se basen en tecnologías asociadas a las cadenas de bloques y los contratos inteligentes, el presente TEG puede sirve de guía para atacar diferentes casos de uso asociados a los enfoques seleccionados.
- Dada la arquitectura planteada, se desarrollaron varios módulos de importancia los cuales sirven de utilidad para trabajos futuros que puedan tomar los datos generados por el presente TEG para aplicar diferentes técnicas de analÃtica predictiva e inteligencia artificial.

7.2. Recomendaciones

Para hacer pruebas con este tipo de tecnologías, la mejor opción es utilizar la herramienta Remix, de esta manera, se pueden realizar pruebas de traspasos de fondos sin tener que involucrar valor real de ether dentro de las pruebas. Sin embargo, en caso de querer realizar pruebas en ambientes de producción, se recomienda realizar pruebas de despliegue en las redes de prueba de Ethereum (Rinkeby, Ropsten y Kovan), existen en la red varios sitios que ofrecen Ether gratis en estas redes de prueba con el fin de ayudar a los desarrolladores a probar sus aplicaciones distribuidas.

7.3. Trabajos futuros

Como se contempló en la arquitectura propuesta para este TEG, los datos y la recopilación de los mismos mediante esta herramienta quedan abiertos y estructurados de manera que sea más sencillo para los expertos en el área de minería de datos o tecnologías afines tomar esos datos y aplicar algoritmos de inteligencia artificial sobre ellos y detectar patrones de comportamiento u otros patrones útiles para el estudio de estas herramientas.

Bibliografía

[1] *Account Types, Gas, and Transactions.* http://ethdocs.org/en/latest/contracts-and-transactions/account-types-gas-and-transactions.html

[2] *Agile.* http://www.ambysoft.com/unifiedprocess/agileUP.html

[3] *Base de datos no relacionales.* http://nosql-database.org/.

[4] *Cassandra.* http://cassandra.apache.org/doc/latest/

[5] *CouchDB.* http://docs.couchdb.org/en/2.1.1/

[6] *Document Object Model (DOM).* https://www.w3.org/DOM/#what

[7] *Dynamo.* https://aws.amazon.com/es/documentation/dynamodb/

[8] *Ethereum Virtual Machine.* http://ethdocs.org/en/latest/introduction/what-is-ethereum.html

[9] *Ganache.* https://github.com/trufflesuite/ganache-core

[10] *HBase.* https://hbase.apache.org/book.html

[11] *InfiniteGraph.* http://www.objectivity.com/products/infinitegraph/

[12] *Infura.* https://infura.io/

[13] *Javascript.* https://developer.mozilla.org/es/docs/Web/JavaScript

[14] *Lenguajes de programacón.* http://www.lenguajes-de-programacion.com/lenguajes-de-programacion.shtml.

[15] *MetaMask.* https://github.com/MetaMask/metamask-extension

[16] *Mocha.* https://mochajs.org/

[17] *MongoDB.* https://www.mongodb.com/what-is-mongodb

[18] *Neo4j.* https://neo4j.com/docs/developer-manual/current/

[19] *Next.* https://github.com/zeit/next.js/

[20] *React*. https://reactjs.org/

[21] *Redis*. https://redis.io/documentation

[22] *Remix*. https://remix.readthedocs.io/en/latest/

[23] *RUP*. http://www.usmp.edu.pe/publicaciones/boletin/fia/info49/articulos/RUP%20vs.%20XP.pdf

[24] *Semantic*. https://react.semantic-ui.com/

[25] *Smart Contracts: 12 Use Cases for Business and Beyond*. https://gallery.mailchimp.com/a87f67248663abe55ad9325d6/files/Smart_Contracts_12_Use_Cases_for_Business_Beyond.pdf

[26] *SolC*. https://github.com/ethereum/solc-js

[27] *Solidity*. https://solidity.readthedocs.io/en/latest/index.html#

[28] *The General Theory of Decentralized Applications, Dapps*. https://github.com/DavidJohnstonCEO/DecentralizedApplications

[29] *Web3*. https://github.com/ethereum/web3.js/

[30] *What is Ether?* https://www.ethereum.org/ether

[31] Back, Adam: Hashcash - A Denial of Service Counter-Measure, August 1, 2002

[32] Buterin, Vitalik: A Next-Generation Smart Contract and Decentralized Application Platform, 2015

[33] Chohan, Usman W.: Cryptocurrencies: A Brief Thematic Review, August 4, 2017

[34] Christopher D. Clack, Lee B.: Smart Contract Templates: foundations, design landscape and research directions, August 4, 2016

[35] Gupta, Manav: Blockchain for Dummies, 2017

[36] Josh Benaloh, Michael de M.: One-way accumulators: a descentralized alternative to digital signatures, 1993

[37] L, Molina: Data mining: torturando los datos., 2016

[38] Li Deng, Dong Y.: Deep Learning Methods and Applications, 2014

[39] Muhammad Saqib Niaz, Gunter S.: Merkle Hash Tree based Techniques for Data Integrity of Outsourced Data, 2015

[40] Nakamoto, Satoshi: Bitcoin: A Peer-to-Peer Electronic Cash System, 2008

[41] Nyce, Charles: Predictive Analytics White Paper, 2007

[42] Saarenvirta, Gary: Machine Learning in Retail, 2010

[43] Steinmetz, Wehrle K.: What Is This Peer-to-Peer About?

[44] Stuart Haber, Scott S.: How to Write Time-Stamp a Digita Document