MECANISMO DE CONSENSO PROOF-OF-REPUTATION PARA SISTEMAS DE BLOCKCHAIN

PROOF-OF-REPUTATION CONSENSUS MECHANISM FOR BLOCKCHAIN SYSTEMS

TRABAJO FIN DE GRADO
CURSO 2022-2023

AUTOR
DANIEL LÓPEZ MARQUÉS

DIRECTOR
JESÚS CORREAS FERNÁNDEZ

GRADO EN INGENIERÍA INFORMÁTICA
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

MECANISMO DE CONSENSO PROOF-OF-REPUTATION PARA SISTEMAS DE BLOCKCHAIN

PROOF-OF-REPUTATION CONSENSUS MECHANISM FOR BLOCKCHAIN SYSTEMS

TRABAJO DE FIN DE GRADO EN INGENIERÍA INFORMÁTICA

AUTOR
DANIEL LÓPEZ MARQUÉS

DIRECTOR
JESÚS CORREAS FERNÁNDEZ

CONVOCATORIA: FEBRERO/JUNIO/SEPTIEMBRE 2023

GRADO EN INGENIERÍA INFORMÁTICA
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

05 DE JUNIO DE 2023

RESUMEN

Blockchain es una tecnología de registros distribuida y descentralizada que permite la creación de registros públicos e inmutables de transacciones y datos, sin necesidad de intermediarios, almacenándolos en "bloques" que forman una cadena que está asegurada mediante técnicas criptográficas. Es descentralizada porque se basa en la idea de que todos los nodos de la red tienen una copia de la cadena de bloques completa. Todos los nodos verifican los bloques que se van añadiendo a la cadena. Para producir estos bloques, todos los nodos deben llegar a un acuerdo para decidir cuál de ellos produce el siguiente bloque. Este procedimiento es lo que se conoce como "mecanismo de consenso". En la actualidad hay varios mecanismos de consenso, los más populares son *Proof of Work* (Bitcoin), *Proof of Stake* (Ethereum) y *Proof of Authority* (redes privadas).

Existen diversos artículos académicos que hablan sobre un mecanismo alternativo llamado "*Proof of Reputation*", que se basaría en la reputación de los diferentes nodos en la red para realizar la selección de validadores de bloques.

El objetivo de este Trabajo de Fin de Grado es diseñar e implementar un mecanismo de consenso basado en reputación partiendo de una implementación de Ethereum ya existente.

Palabras clave

Blockchain, Mecanismos de consenso, Sistemas de reputación, Ethereum, Hyperledger Besu, Mecanismo de consenso Clique

ABSTRACT

PROOF-OF-REPUTATION CONSENSUS MECHANISM FOR BLOCKCHAIN SYSTEMS

Blockchain is a distributed and decentralized record-keeping technology that allows the creation of public and immutable records of transactions and data, without the need for intermediaries, by storing them in "blocks" that form a chain that is secured by cryptographic techniques. It is decentralized because it is based on the idea that all nodes in the network have a copy of the complete blockchain. All nodes verify the blocks that are added to the chain. To produce these blocks, all nodes must reach an agreement to decide which of them produces the next block. This procedure is known as a "consensus mechanism". There are currently several consensus mechanisms, the most popular being Proof of Work (Bitcoin), Proof of Stake (Ethereum) and Proof of Authority (private networks).

There are several academic articles that talk about an alternative mechanism called "Proof of Reputation", which would be based on the reputation of the different nodes in the network to make the selection of block validators.

The aim of this thesis is to design and implement a reputation-based consensus mechanism based on an existing Ethereum implementation.

Keywords

Blockchain, Consensus Mechanisms, Reputation Systems, Ethereum, Hyperledger Besu, Clique Consensus Mechanism

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

Capítulo 1 - Introducción

1.1 Antecedentes

Blockchain, o cadena de bloques, es una tecnología que permite crear un registro público y seguro de transacciones y datos, de forma descentralizada y distribuida, almacenándolos en "bloques" que contienen una referencia hash del bloque anterior, formando así una cadena inmutable. (Este concepto se explicará de forma más detallada en la sección 2). (1) (2) (3)

La inmutabilidad de los datos almacenados en los bloques de la cadena se garantiza gracias al uso de funciones hash criptográficas, ya que una función hash es un algoritmo que produce una serie de caracteres para un input, y que, si se modifica el input, produce una serie totalmente diferente. Gracias a esta característica, en caso de modificarse la información de los bloques, se detectaría fácilmente.

Además de esta característica de inmutabilidades, *blockchain* también garantiza la autenticidad de los datos almacenados, ya que cada una de las transacciones realizadas se firma digitalmente por los emisores utilizando claves criptográficas asimétricas. De esta forma, se garantiza que no se podrá suplantar al emisor ni alterar las transacciones una vez realizadas.

Sin embargo, el sistema *blockchain*, por ser una red distribuida y descentralizada, requiere de un mecanismo que permita a los diferentes nodos de la red colaborar y mantenerse seguros. Para ello, todos los nodos de la red deben ponerse de acuerdo sobre quién debe producir el siguiente bloque y la validez de su contenido. Esto se conoce como mecanismo de consenso, y se utiliza para producir nuevos bloques en la cadena de bloques, es decir, todos los nodos de la red deben ponerse de acuerdo sobre quién debe producir el siguiente bloque.

En la actualidad existen múltiples mecanismos de consenso, siendo los más conocidos Proof of Work (de la red Bitcoin), Proof of Stake (de la red Ethereum) y Proof of Authority (de redes privadas). Cada uno de ellos utiliza diferentes conceptos y mecanismos para alcanzar el consenso entre los nodos, pero la finalidad es la misma: seleccionar de forma consensuada qué nodo producirá el siguiente bloque.

Este trabajo se basa en la idea de crear un prototipo de mecanismo de consenso de redes *Blockchain* innovador, teniendo como idea central los sistemas basados en reputación propuestos en diversos artículos académicos (4) (5) (6). Un mecanismo de consenso basado en sistemas de reputación permitiría ajustar de forma más dinámica que otros mecanismos los criterios utilizados para elegir el siguiente validador (nodo que producirá el siguiente bloque). Podría utilizarse en redes privadas o permisionadas, o incluso en redes mayoritarias como Bitcoin o Ethereum.

Como punto de partida se puede tomar un nodo Ethereum ya existente, como Geth (7), programado en Go, o Hyperledger Besu (8), programado en Java, que permiten la creación de redes *Blockchain* privadas, ideales para hacer pruebas con este tipo de mecanismo de consenso.

1.2 Objetivos

El objetivo principal de este trabajo consiste en estudiar las características de un sistema de *blockchain* que utilice un mecanismo de consenso basado en *Proof of Reputation* (PoR). Para ello, se va a implementar un prototipo que nos permita evaluar los aspectos más relevantes del sistema, utilizando como punto de partida una implementación de código abierto de Ethereum existente. Concretamente, los objetivos específicos de este trabajo son los siguientes:

- Estudiar mecanismos de consenso alternativos a los más populares, centrándose especialmente en *Proof of Reputation*.
- Estudiar la posibilidad de implementar un mecanismo de consenso programado en un *Smart Contract*, delegando en él toda la gestión del mecanismo y liberando a los nodos de esa funcionalidad. Esto permitiría tener un mecanismo de consenso seguro, transparente, trazable y actualizable.
- Estudiar la aplicabilidad de sistemas de reputación a los mecanismos de consenso de sistemas *blockchain*. Diseñar un mecanismo de consenso PoR (*Proof of Reputation*).

- Estudiar las implementaciones actuales de sistemas de *blockchain* de Ethereum para elegir la más adecuada para la implementación de un prototipo de consenso PoR.
- Una vez implementado el prototipo, hacer pruebas montando una red de nodos local, y ver hasta qué punto sería escalable en una red real.

1.3 Plan de trabajo

Durante el desarrollo de este trabajo se han mantenido reuniones semanales entre tutor y alumno para el seguimiento y planificación de las siguientes tareas:

1. Estudiar las implementaciones de Ethereum existentes y decidir cuál es la más adecuada para utilizar como base de este trabajo.
2. Estudiar los mecanismos de consenso existentes en el nodo y ver cuál sería más similar a *Proof of Reputation*.
3. Estudiar en detalle las propuestas de consenso basadas en *Proof of Reputation* (PoR) publicadas en artículos académicos.
4. Montar una red de *blockchain* de pruebas con uno o dos nodos para conocer el funcionamiento de la implementación de Ethereum elegida y compilar y ejecutar el código fuente de esa implementación.
5. Diseñar un esquema básico de PoR a partir de las propuestas académicas estudiadas en el punto 3. Evaluar la viabilidad de la utilización de *Smart Contracts* para implementar el mecanismo de consenso PoR.
6. Implementar el diseño, ya sea con un *Smart Contract* o en el programa de gestión del propio nodo, según se haya diseñado en el paso anterior.
7. Probar el prototipo implementado para poder compararlo con los mecanismos existentes y hacer pruebas de escalabilidad.
8. Realizar la documentación del proyecto encuadrando el mecanismo alternativo elegido en los algoritmos ya existentes.

Capítulo 2 - Introducción a la tecnología *Blockchain*

En este capítulo se incluye una breve introducción a los conceptos más importantes de la tecnología *blockchain* y la cadena de bloques.

Blockchain es una tecnología de registro distribuido que permite la creación de un registro público y seguro de transacciones y datos, sin necesidad de intermediarios. Las transacciones y los datos se almacenan en bloques que van formando una cadena denominada *blockchain* (1) Esta tecnología se ha popularizado en los últimos años gracias a sus características, que aportan transparencia, seguridad y descentralización a diversas industrias.

La principal característica de *Blockchain* es la inmutabilidad de los datos almacenados en sus bloques, gracias al uso de funciones de hash criptográficas. Cada bloque contiene una referencia al bloque anterior a través de un hash criptográfico, que es una representación única y firmada digitalmente de los datos contenidos en el bloque, como se muestra en la figura 2-1. Si se realizara cualquier cambio en los datos de uno de los bloques, el hash generado también cambiaría, y por lo tanto la referencia almacenada en el siguiente bloque no coincidiría, y se invalidaría la cadena.

Gracias a esta característica, cualquier intento fraudulento de modificar un bloque ya existente en la cadena sería detectado fácilmente, porque el hash del bloque alterado no coincidiría con el hash almacenado en el siguiente bloque. Esta característica de inmutabilidad garantiza la integridad de los datos almacenados en la *blockchain* y es la base de la confianza de todos los participantes de la red.

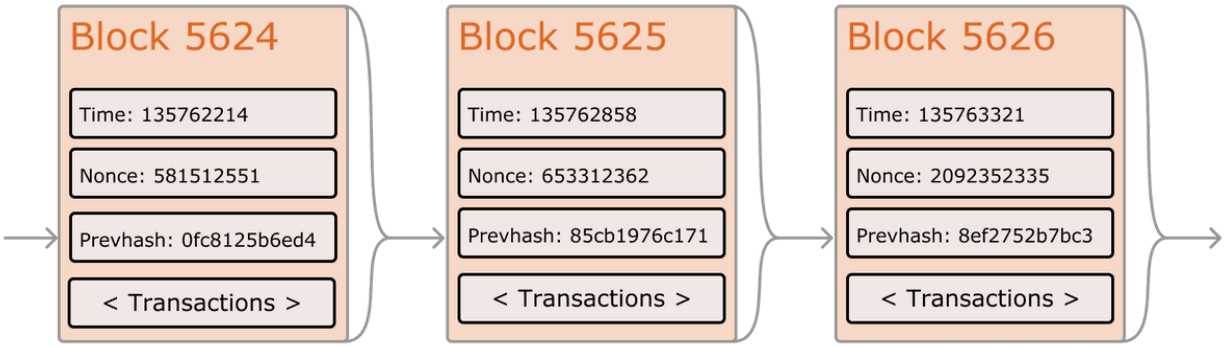

Figura 2-1 Representación gráfica de la cadena de bloques. Fuente: Ethereum Whitepaper (2)

Además de la inmutabilidad de los datos ya almacenados en la cadena de bloques, existe otra capa de seguridad que consiste en la firma criptográfica de las transacciones. En cada transacción que se realiza en la *blockchain*, los emisores utilizan claves criptográficas asimétricas para firmar digitalmente la transacción. Por lo tanto, cada transacción está asociada a una clave pública única y su correspondiente clave privada, de forma que solo el titular de esta puede generar una firma válida. La firma criptográfica de las transacciones asegura que no se pueda suplantar al emisor ni alterarlas una vez que se han emitido y, por lo tanto, garantiza la autenticidad de los datos en la cadena de bloques. (2) (3)

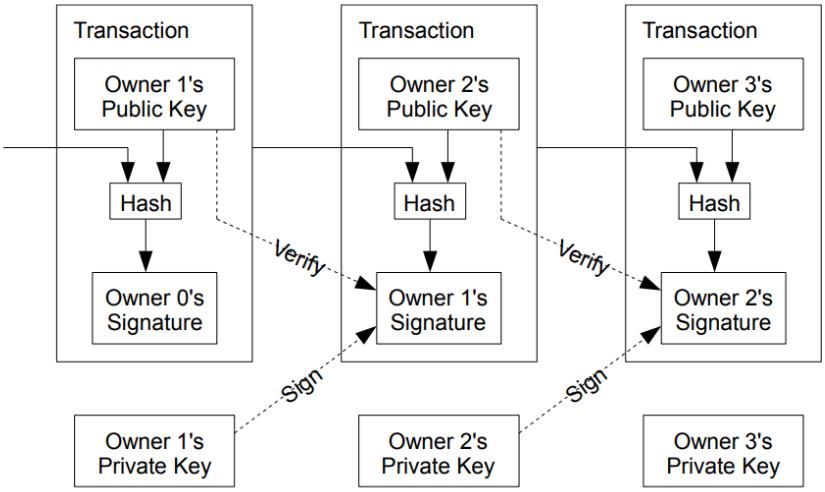

Figura 2-2 Representación gráfica de la firma de transacciones en la blockchain. Fuente: Bitcoin Whitepaper (3)

La cadena de bloques que almacena datos es compartida por cada uno de los nodos de la red, es decir, cada nodo que se conecte a la *blockchain* tendrá una copia de la cadena completa en su equipo local, y cada vez que un nuevo bloque es añadido por un nodo especial (un "minero", o "validador"), se redistribuye al resto de los nodos de la red para que todos completen su copia de la cadena de bloques.

De esta forma, es fácil detectar si ha habido alguna manipulación en los datos anteriores de la cadena, pues todos los nodos tienen una copia.

Para asegurar la seguridad e integridad de las redes *blockchain*, se utilizan los llamados mecanismo de consenso, que son procedimientos que sirven para que todos

los nodos de la red se pongan de acuerdo para crear nuevos bloques y validar los actuales, sin necesidad de ningún tercero (descentralización).

El primer mecanismo de consenso, y más conocido, fue *Proof of Work* (Prueba de Trabajo), actualmente utilizado por Bitcoin y redes minoritarias. Otro mecanismo popular es Proof of Stake (Prueba de Participación), que la utiliza Ethereum, la segunda *blockchain m*ás conocida. Y el tercer mecanismo más conocido, aunque este se utiliza mayoritariamente en redes privadas, es *Proof of Authority* (Prueba de Autoridad).

Capítulo 3 - Estado del arte

Este capítulo se compone de dos secciones principales. Primero se introduce la noción de mecanismo de consenso en los sistemas de *blockchain* y se describen los sistemas más relevantes. A continuación, en la sección 3.2, se introducen los aspectos más importantes de las propuestas existentes de sistemas de reputación utilizados en mecanismos de consenso de sistemas de *blockchain*.

3.1 Mecanismos de consenso

Como se ha introducido en el apartado anterior, *Blockchain* funciona mediante diferentes mecanismos de consenso, que se basan en llegar a un consenso entre todos los nodos de la red para producir un nuevo bloque, y añadirlo a la cadena de bloques. Llegar a un consenso significa que al menos el 66% de los nodos de la red coinciden en el siguiente estado general de la red (9).

Es importante destacar que estos mecanismos de consenso son fundamentales para las redes *Blockchain*, por ser sistemas descentralizados y distribuidos en las que no existe un nodo central o servidor que tenga el control de la información y las decisiones de la red. Los mecanismos de consenso permiten a estos sistemas colaborar entre los diferentes nodos y mantenerse seguros. (9)

En redes descentralizadas todos los nodos tienen el mismo peso e importancia para la toma de decisiones, por lo tanto, todos ellos deben ponerse de acuerdo sobre el estado actual y las modificaciones de la cadena de bloques, y, si un nodo actúa de forma maliciosa sería fácilmente detectado por los demás y sería expulsado. Si no existiese un mecanismo de consenso para garantizar esto, cada nodo podría tomar decisiones propias, posiblemente maliciosas, y modificar la información almacenada en la cadena de bloques, causando graves problemas de seguridad e integridad en la red.

Se planteó inicialmente el uso de "algoritmos" de consenso tolerantes a fallos bizantinos (Problema de los generales bizantinos (10)), algoritmos muy complejos. Sin embargo, finalmente se descartaron y se crearon los actuales **mecanismos** de consenso.

Si bien los mecanismos de consenso son importantes para evitar vulnerabilidades en los sistemas *blockchain* (por ejemplo, toma de decisiones unilaterales por parte de algún nodo), también pueden ocasionar problemas de seguridad en redes minoritarias, en las que hay pocos nodos. Este es el caso del ataque del 51%, que puede tener lugar si un grupo de nodos malintencionados llegan a controlar más del 50% de los nodos de la red, y les permitiría controlar la cadena y realizar transacciones fraudulentas. Este tipo de ataques han ocurridos en redes reales, como en la *blockchain* de Bitcoin Gold en 2018 (11). Sin embargo, las redes mayoritarias como Bitcoin o Ethereum no han sufrido ataques de este tipo que hayan tenido éxito, ya que la magnitud de estas hace casi imposible que se llegue a controlar el 51% de la red.

Estos mecanismos de consenso evitan determinadas vulnerabilidades en este tipo de sistemas, ya que garantizan que todos los nodos están de acuerdo, y si un nodo actúa de forma maliciosa sería fácilmente detectado por los demás y sería expulsado.

En el caso de Bitcoin, se utiliza *Proof of Work*. En este mecanismo de consenso, cada nodo de la red debe probar que ha invertido recursos computacionales para producir el siguiente bloque de la cadena. Los nodos que realizan este trabajo, también llamado "minado", se conocen como "mineros" (12). Para realizar el minado, los mineros deben resolver un algoritmo matemático complejo, que consiste en encontrar un valor de *nonce* (se explicará a continuación) para el cual el hash del bloque es más pequeño que un valor específico. El *nonce* de un bloque es un número contenido dentro del mismo, que por su nombre en inglés "nonce" significa "número usado solo una vez". El primero en resolverlo añadirá el bloque a la cadena y recibirá una recompensa. Como puede haber más de un bloque que se produzca simultáneamente, se puede producir ramificaciones de la cadena. Para decidir cuál es la rama válida, este mecanismo de consenso determina que la subcadena más larga es la seleccionada, ya que es la que mayores recursos ha utilizado. (ver figura 3-1)

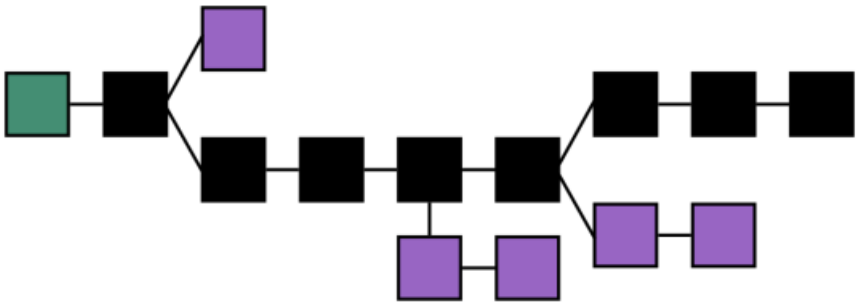

Figura 3-1 Ramificaciones de una cadena de Bitcoin. Fuente: (23)

Este mecanismo ha sido ampliamente criticado por sus problemas medioambientales de consumo de energía excesivo, ya que la única forma de resolver el algoritmo es mediante prueba y error. Se estima que Bitcoin consume más energía que el país Noruega entero, un total del 0.65% del consumo mundial de energía. (13)

A raíz de esa polémica, empezaron a surgir otros mecanismos de consenso, como es el caso de *Proof of Stake*, recientemente incorporado a Ethereum, la segunda criptomoneda más popular. Este mecanismo desecha la idea de resolver algoritmos matemáticos y otorga la posibilidad de producir un bloque a un nodo en función de la cantidad de dinero que haya aportado como fianza (*stake*) para la validación de bloques y también un factor de aleatoriedad. De esta forma se resuelve el problema medioambiental, ya que este mecanismo de consenso tiene un consumo de energía ínfimo.

También existen otros mecanismos de consenso minoritarios utilizados, sobre todo, en redes privadas, como *Proof Of Authority*, que requiere que los nodos sean validados por entidades reales de confianza para poder ser elegidos validadores de bloques.

Para el desarrollo de este trabajo se ha partido de Hyperledger Besu y de su mecanismo de consenso Cliqué, habiendo tenido en cuenta para esta decisión su sencillez y documentación en la página web de Hyperledger Besu. En la sección 4 se indicarán las razones por las que se ha elegido esta implementación.

Hyperledger Besu es un *software open-source* que permite a un nodo formar parte de una red con tecnología Ethereum. Existen múltiples redes de *blockchain* que

utilizan la tecnología Ethereum, de las que destaca la *mainnet* de Ethereum, que es la red pública que soporta la criptomoneda Ether, así como muchas otras monedas basadas en tokens que se ejecutan como contratos dentro de esta red. Además, Hyperledger Besu se puede utilizar para crear redes privadas basadas en esta misma tecnología utilizando uno o varios nodos que se comunican entre sí. Los mecanismos de consenso que soporta por defecto, y que son seleccionables para crear una red privada, son Cliqué, QBFT, e IBFT 2.0 (basados en *Proof of Authority*), *Proof of Stake*, y *Etash* (*Proof of Work*).

3.2 Introducción a los sistemas de reputación

Casi desde el inicio de los sistemas de *blockchain*, se han planteado propuestas tanto académicas como industriales relativas a nuevos mecanismos de consenso, que permitiesen garantizar que los nodos elegidos como validadores fuesen fiables y que también la selección de estos fuese justa y transparente. Este es el caso de los sistemas basados en reputación.

En particular, los mecanismos de consenso basados en sistemas de reputación han sido propuestos en varias contribuciones académicas. Entre ellos, en esta sección vamos a analizar las propuestas publicadas por Aluko y Kolonin (4), Do et al. (5) y Zhuang et al. (6). En el whitepaper de Ethereum en 2014 (2) ya se mencionaba también la posibilidad de añadir sistemas de reputación de forma sencilla, utilizando *Smart Contracts*.

Estos sistemas se basan en las reputaciones de los diferentes nodos de la red como factor de selección para ser elegidos como nuevos validadores o productores de bloques, y plantean diferentes sistemas de votación. Sin embargo, La forma de calcular dicha reputación y su uso varía según el enfoque, así como el sistema de votación.

En el artículo de Zhuang et al. (6), el nodo con mayor reputación de la red es el que produce un bloque, y el 20% de los nodos con mayor reputación votan por ese bloque para validar que es correcto y añadirlo a la red.

Para el cálculo de la reputación utiliza tres factores, como se muestra en la figura 3-2: la antigüedad de su dinero, es decir, cuánto tiempo lleva su dinero en la cartera del nodo; la actividad social del nodo, es decir, la cantidad de transacciones que ha

realizado; y la contribución al mecanismo de consenso, es decir, si ha sido validador, cuantos bloques ha producido en total.

El sistema de votación en este caso solo es utilizado para validar los bloques que se van a producir, pero no para seleccionar el nodo con mayor reputación.

$$R_i = \beta_1 R_s(S_i, t) + \beta_2 R_a(A, V) + \beta_3 R_c(N_i) \qquad (2)$$

where

$$R_s(S_i, t) = \alpha \log(S_i t) \qquad (3)$$

$$R_a(A, V) = \sum_{k=1}^{j} A_k \log(V_k) S(k) \qquad (4)$$

$$R_c(N_i) = \gamma_1 N_{ic} T_{total} - \gamma_2 N_{ie} T_{total} \qquad (5)$$

Where β_i is the weight. R_s denotes the value of reputation converted by the age of currency. R_a denotes the degree of social activity of a node. R_c denotes the contribution of consensus.

Figura 3-2 Esquema de reputación descrito en el artículo de Zhuang et al. (6)

En el artículo de Do et al. (5), la reputación se calcula también en base a tres factores, como se muestra en la figura 3-3: El *stake power*, es decir, la cantidad de dinero en *stake*; el uso de recursos del nodo, es decir, la cantidad de recursos (CPU, RAM, banda ancha) que utiliza; y la cantidad de transacciones de la cuenta, es decir, a mayor cantidad de transacciones tenga ese nodo, tanto de entrada como de salida, mayor será este factor.

Sin embargo, este artículo plantea un *Delegated Proof of Reputation*, basado en *Delegated Proof of Stake*, que es una modificación del PoS básico, que se basa en votar a un grupo de nodos que se encarguen de velar por el correcto funcionamiento del mecanismo de consenso, y entre ellos se selecciona aleatoriamente el productor de bloque en cada ronda.

Es por esto por lo que en este artículo el sistema de votación es diferente al artículo de Zhuang et al. (6), tanto en forma como en propósito. Es diferente en forma,

porque en este caso los votos son ponderados por la reputación de cada nodo, es decir, a mayor reputación mayor valor tiene su voto. Y también es diferente en propósito, porque el objetivo de esta votación es elegir a un grupo de nodos "delegados", al igual que en DPoS, para que estos nodos sean los validadores hasta la siguiente ronda de votación. De este grupo de nodos delegados, se elige el validador de bloque en cada turno de forma aleatoria.

$$\textbf{\textit{Vote weight}} = \textbf{\textit{Reputation score}} = \boldsymbol{Rep} = \boldsymbol{f(P,U,R)}$$

$$\boldsymbol{Rep} = \boldsymbol{c_1 P + c_2 U + c_3 R}$$

$\boldsymbol{P}$ presents *stake power* of an account computed based-on the number of staked coins.

$\boldsymbol{U}$ indicates (CPU, RAM, bandwidth) *resource usage* of an account. Such infrastructure is indispensable regarding any internet-based network, hence it should be evaluated.

$\boldsymbol{R}$ is the most significant factor in our innovation. It stands for *transaction-based reputation rating* of an account. We appreciate active accounts with many (in/out) transactions. Like web-hyperlinks, transactions can be considered as a useful indicator for performance of an account and the entire network as well. It

Figura 3-3 Esquema de reputación descrito en el artículo de Do et al. (5)

En cuanto al artículo de Aluko y Kolonin (4), el enfoque es similar al artículo de Do et al. (5), con la diferencia de que la reputación se calcula en base a un sistema de *ratings*, es decir, cada nodo puede puntuar a otro y ser puntuado después de una transacción, para asignarle una reputación cuanto mayor puntuación se le haya asignado.

También describe un sistema de votación basado en votos ponderados por la reputación del nodo votante, y se elige a un *consensus group*, es decir, un grupo de validadores de los que, en cada ronda, se selecciona aleatoriamente al líder que producirá el siguiente bloque. Este grupo se mantendrá hasta la siguiente ronda de votación. Se muestra este procedimiento en la figura 3-4.

Algorithm 2: Leader Selection

Result: Leader L for the round k
input: set of highest ranking nodes G;
1: leader = random(G);
2: broadcastLeader($leader$);

Figura 3-4 Esquema de selección de nodo líder descrito en el artículo de Aluko y Kolonin (4)

A partir de las propuestas de artículos académicos que se han descrito, se diseñó el esquema de reputación a ser implementado en este trabajo. Este esquema de reputación será descrito en la sección 5.3.

Capítulo 4 - Arquitectura de Hyperledger Besu

Los programas que se utilizan en los nodos de sistemas de *blockchain* se llaman "clientes". Al principio de este trabajo se decidió que el sistema de *blockchain* más conveniente para desarrollar el prototipo fuera Ethereum porque implementa un sistema de *blockchain* programable mediante contratos inteligentes y tiene un gran número de herramientas de código abierto disponibles. En particular, existen diversas implementaciones de clientes de Ethereum que se pueden utilizar gratuitamente y con el código fuente disponible. De hecho, dos de los clientes de Ethereum más importantes son Geth y Hyperledger Besu.

Geth (7) está desarrollado en el lenguaje de programación Go y es con diferencia el cliente de Ethereum más utilizado por los nodos de la *Ethereum mainnet* (red pública principal basada en Ethereum que sirve de soporte a la criptomoneda Ether). En la figura 4-1 se puede ver el porcentaje de su utilización en la red principal de Ethereum:

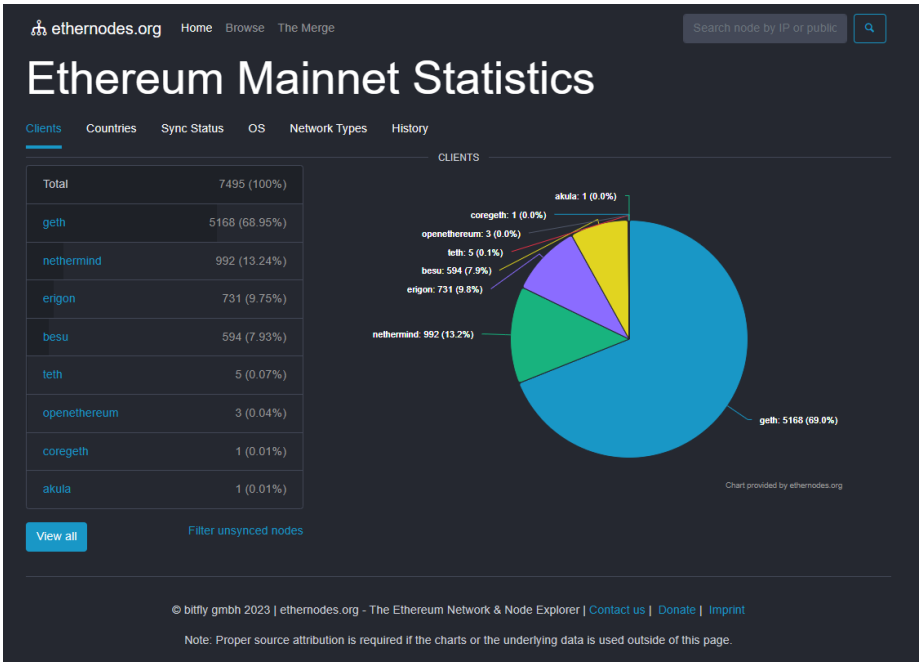

Figura 4-1 Fuente: https://www.ethernodes.org/

Por su parte, Hyperledger Besu está desarrollado en Java y ha sido desarrollado por la Hyperledger Foundation (8), fundación que se encarga de toda una familia de

sistemas de *blockchain* de código abierto patrocinada por la Linux Foundation (14). Aunque hay un número de nodos de la red principal de Ethereum que utilizan Hyperledger Besu, este cliente de Ethereum se utiliza fundamentalmente para redes de *blockchain* permisionadas, en las que el acceso a la red no es público, sino que se realiza utilizando control de identidad mediante credenciales. Después de estudiar las características de estos y otros clientes de Ethereum, finalmente se optó por Hyperledger Besu, principalmente por el lenguaje de programación en el que está desarrollado, del que se disponen amplios conocimientos, y por disponer de documentación de desarrollo detallada y fácilmente accesible.

En segundo lugar, se analizó el código y la arquitectura de Hyperledger Besu para estudiar la viabilidad de utilizarlo como base de la que partir para implementar el mecanismo de *Proof of Reputation*. Siguiendo la documentación, se consiguió compilar el código fuente usando *Gradle*, y ejecutarlo con terminales en Windows. En principio hubo un problema que impedía utilizar la API JSON-RPC para consultar datos de la *blockchain* cuando estaba en ejecución, finalmente se consiguió solucionar usando la herramienta *Postman*, que facilita las llamadas a la API.

Hyperledger Besu se organiza en una serie de carpetas (ver figura 4-2) que contienen los diferentes componentes de la arquitectura. En este enlace se encuentra el repositorio de la versión 22.10.3, de la que se ha partido como base:

https://github.com/hyperledger/besu/tree/5161b613a059fb2c1fc11412a02c1701ee52aa72

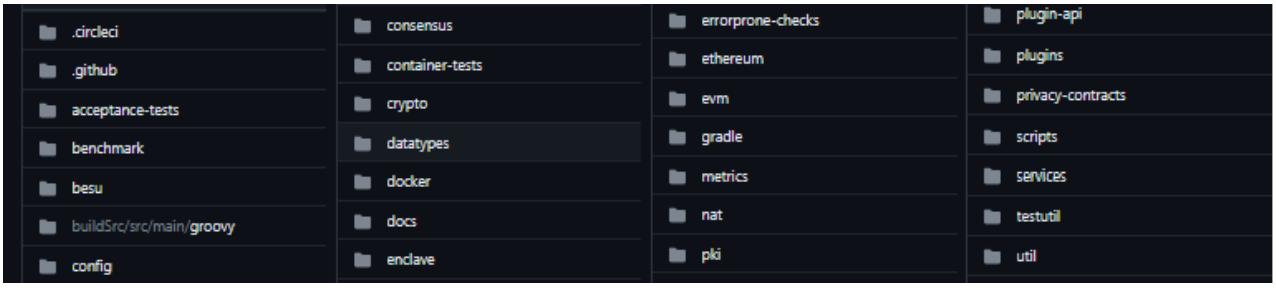

Figura 4-2 Arbol de directorios de Hyperledger Besu

Los directorios **besu** y **ethereum** contienen la mayor parte de la funcionalidad común de todos los mecanismos de consenso de Hyperledger Besu. Por su parte, el directorio **config** contiene las clases relacionadas con la configuración del cliente. En

cuanto al directorio **crypto** contiene las clases que definen el modelo de datos para la seguridad del cliente (por ejemplo, *publicKey* y *privateKey*). Los directorios **docker, evm** y **services** están relacionado con el despliegue de la aplicación. Por su parte, los directorios **acceptance-test**, **container-tests,** y **testutil** contienen diferentes tipos de pruebas para comprobar el correcto funcionamiento del código del cliente.

En general, la mayoría de los ficheros de cada mecanismo de consenso se ubican en una carpeta dentro de la carpeta **consensus**. Cada mecanismo tiene su propia carpeta con sus clases que implementan la funcionalidad específica del mismo.

Por otra parte, al igual que en otras redes *blockchain*, cuando Hyperledger Besu se utiliza para crear una cadena de bloques privada, debe desplegar un bloque génesis, que es el primer bloque de la cadena. Este bloque es un bloque especial que no tiene un bloque que le preceda, y por ello necesita de una serie de parámetros iniciales que definan la configuración de la cadena en los siguientes bloques.

Para desplegar el **bloque génesis**, Hyperledger Besu utiliza un fichero cuyo nombre se especifica en el comando de ejecución. En él se pueden especificar propiedades como el ID de la cadena de bloques, el mecanismo de consenso que se quiere utilizar, propiedades específicas del mecanismo (por ejemplo, el tiempo entre bloques), el límite de gas, el *nonce* del bloque inicial, el *timestamp* inicial, y reservar *addresses* para nodos (clientes de la red), pre-asignándoles un balance; así como *addresses* para *Smart Contracts* pre-desplegados en el bloque génesis, en los que además de especificar su balance se pueden añadir valores al *storage*, como se indica en la documentación oficial (15). (ver figura 4-3).

Figura 4-3 Contenido de un fichero de creación de bloque génesis en Hyperledger Besu

Además, Hyperledger Besu utiliza un sistema de hilos concurrentes, de forma que, cuando se comienza a minar un bloque, se crea un *thread* nuevo, y este no termina hasta que, o bien ha minado el bloque, o bien se ha rechazado por no ser su turno. Pasados 15 segundos (tiempo entre bloques, configurable en el bloque génesis, mencionado anteriormente) un nuevo hilo despierta y comienza a intentar minar un bloque.

En segundo plano se encuentran otros hilos que se encargan de importar los bloques que se reciben de otros nodos, es decir, cuando un nodo produce un bloque hay un hilo en el resto de los nodos que se encarga de importar ese bloque que se acaba de producir y añadirlo a la cadena local de cada bloque, previamente haciendo una serie de validaciones para comprobar que ese bloque se ha validado y producido correctamente y no es malicioso.

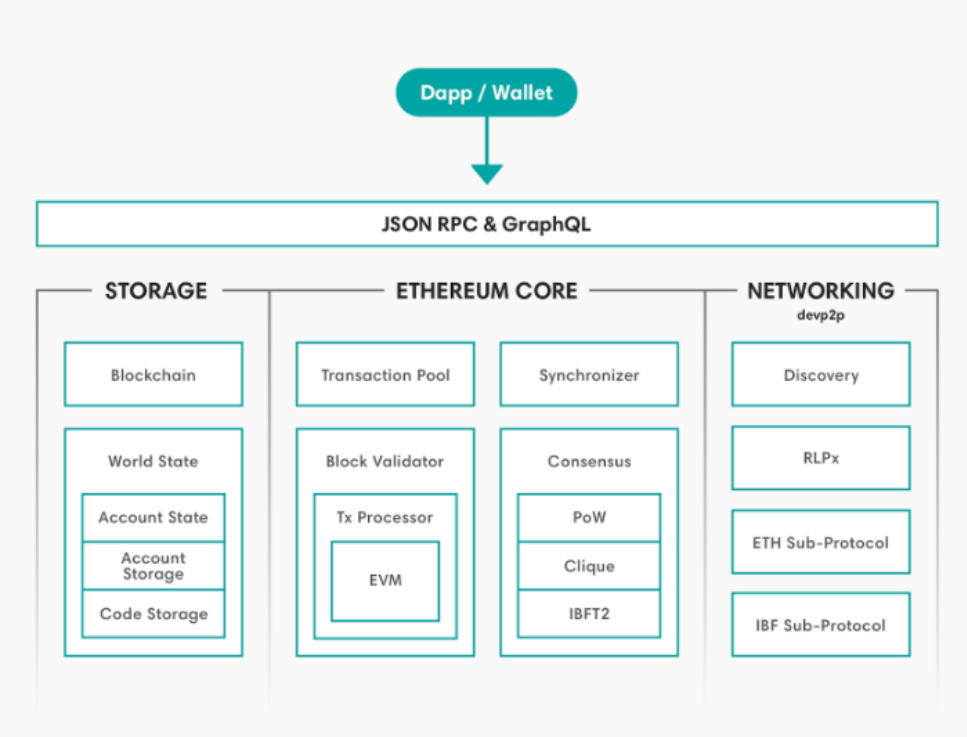

Figura 4-4 Arquitectura de Hyperledger Besu. Fuente: (24)

En la figura 4-4 se muestra en un diagrama los diferentes componentes de la arquitectura de Hyperledger Besu. El componente "Consensus" que está en el centro a la derecha representa la carpeta consensus, con los diferentes mecanismos de consenso disponibles en Hyperledger Besu por defecto.

Capítulo 5 - Diseño del prototipo

Este capítulo está dividido en tres secciones. Primero se va a describir el esquema general que se ha elegido para diseñar el prototipo. A continuación, se describirá la arquitectura que se ha diseñado, y por último se describirá el esquema de reputación elegido, es decir, qué factores se han decidido utilizar para el cálculo del valor de reputación de cada nodo, en base a las ideas recopiladas de diferentes artículos académicos.

5.1 Esquema general

La idea que se planteó al inicio del proyecto para implementar un mecanismo de consenso basado en reputación para Hyperledger Besu consistía en utilizar como punto de partida uno de los mecanismos de consenso existentes en la distribución de Besu (en particular el mecanismo Clique) y crear sobre esta estructura base el nuevo mecanismo implementado directamente en Java y compilarlo junto con el resto de Hyperledger Besu. El inconveniente principal de este planteamiento es que es difícil actualizar las características del sistema de reputación.

Por este motivo, finalmente se decidió estudiar la posibilidad de implementar el mecanismo de consenso mediante un S*mart Contract* que está instalado en la propia *blockchain*. Este enfoque proporciona una serie de ventajas:

- Seguridad: El código ejecutable del mecanismo de consenso está instalado en la propia red de *blockchain*, de forma que la propia *blockchain* garantiza que no se ha suplantado el código de forma fraudulenta, por ejemplo, atacando algunos nodos de la red.

- Transparencia: El código ejecutable del mecanismo de consenso está disponible para cualquiera, que lo puede estudiar y analizar. Además, cualquier usuario de la red de *blockchain* puede ejecutar algunas funciones del contrato para consultar información acerca del mecanismo de consenso, como el siguiente validador, la reputación actual de cada nodo, etc.

- Trazabilidad: Una de las características más relevantes de la tecnología *Blockchain* es que registra todas las operaciones (transacciones) realizadas sobre la red. Si el sistema de reputación del mecanismo de consenso está implementado en un contrato, es posible recuperar todas las acciones realizadas por el mecanismo de consenso.

- Posibilidad de actualización: Aunque los contratos instalados en un sistema de *blockchain* son inmutables y no se pueden actualizar, se puede implementar un mecanismo basado en un contrato proxy que permita actualizar fácilmente el código del contrato con el sistema de reputación. De esta forma, sin necesidad de reinstalar los clientes Hyperledger Besu, se puede actualizar el mecanismo de reputación con cambios que afecten no solo a la parametrización del sistema, sino también a modificaciones más importantes que supongan cambios en el código ejecutable que implementa el sistema de reputación (siempre que se mantenga el mismo interfaz público del contrato). Este aspecto es especialmente interesante para mejoras del sistema o corrección de errores, aunque el diseño del sistema debe ofrecer un nivel de seguridad muy elevado debido al enorme riesgo que supone una actualización indebida del sistema de reputación.

A este mecanismo de consenso basado en sistemas de reputación e implementado mediante un contrato inteligente se ha decidido llamarle "Repu", basándose en los nombres de los mecanismos de consenso existentes en Hyperledger Besu, que tienen nombres cortos.

En la siguiente sección se hablará sobre la arquitectura que se ha diseñado a partir de este esquema general que se plantea.

5.2 Arquitectura del prototipo

En esta sección se describirá la arquitectura del prototipo (representada gráficamente en la figura 5-1), así como el proceso para diseñar e implementar este.

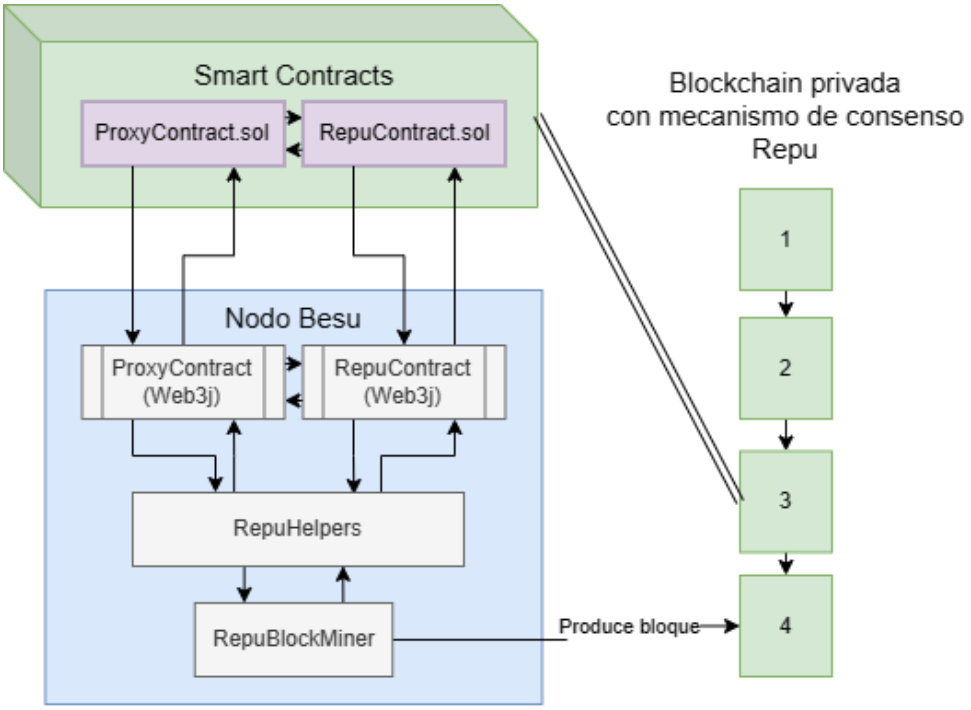

Figura 5-1 Diagrama general de la arquitectura del prototipo

La parte de la derecha, formada por **cuadrados** de color **verde**, representa una cadena de bloques, siendo cada cuadrado un bloque, con su número correspondiente, y las flechas que los unen representan el orden en que se han generado.

En la parte izquierda, el **área superior** de color **verde** representa de forma ampliada el bloque número 3, así como los *Smart Contracts*¸ representados en color morado, que son programados en lenguaje Solidity.

El primero, **ProxyContract**, es un contrato sencillo que implemente la posibilidad de actualizar el contrato del mecanismo de reputación, almacenando su *address* y permitiendo actualizarla con ciertas restricciones por seguridad.

El segundo, **RepuContract**, es el contrato que implementa el mecanismo de consenso basado en reputación, actualizable mediante el contrato Proxy, que implementa el sistema de votación para actualizar la lista de nodos validadores, que

permite calcular la reputación de cada nodo de forma dinámica, y se encarga de decidir quién será el siguiente nodo validador.

Por su parte, el **área inferior** de color **azul** representa el cliente Hyperledger Besu, programado en Java. Cada elemento dentro del bloque azul representa una clase de Java. El cliente Hyperledger Besu está formado por un gran número de clases, en la figura 5-1 solo se representan las clases necesarias para comprender la arquitectura del prototipo. Los dos primeros de la parte de la parte superior, que tienen unas líneas verticales en los laterales, representan las "APIs" de los *Smart Contracts*, es decir, las clases de Java que hacen de puente entre el código funcional del cliente y los contratos inteligentes de la *blockchain*. Estas clases utilizan la librería Web3j (16), que permite trabajar con la cadena de bloques sin tener que implementar un código de integración propio, facilitando así esta tarea.

La clase **ProxyContract** implementa la comunicación entre el cliente Hyperledger Besu y el *Smart Contract* ProxyContract, mientras que la clase **RepuContract** implementa la comunicación con el contrato RepuContact.

En cuanto a la clase **RepuHelpers**, es la clase que implementa la mayor parte de funcionalidad en Java, que hace de intermediaria ente la clase productora de bloques (RepuBlockMiner, que será descrita a continuación) y los *Smart Contracts* (las clases Web3j, descritas anteriormente).

Finalmente, la clase **RepuBlockMiner** se encargar de producir bloques, así como de llamar a métodos de RepuHelpers relacionados con el mecanismo de consenso, como comprobar si es turno de producir bloque para un cierto nodo, o comprobar si es turno de votar.

Las **flechas** que se muestran en la figura representan las interacciones que existen entre los diferentes elementos del sistema.

5.3 Esquema de reputación utilizado

El desarrollo de este TFG se ha basado fundamentalmente en el enfoque de Do et al. (5), aunque para el sistema de votación también se ha utilizado uno de los factores que se proponen en el artículo de Zhuang et al. (6).

El esquema de reputación utilizado se muestra en la figura 5-2.

Vote weight = Reputation = R = c_1B + c_2N + c_3P

c_1, c_2 y **c_3** representan los pesos (*weights*) de cada factor de reputación.

B representa el balance (cantidad de dinero) de un nodo

N representa el *nonce* (número de transacciones) de un nodo

P representa la participación en el consenso (número de bloques producidos)

Figura 5-2 Esquema de reputación implementado en Repu

El primer factor **B** está basado en el primero del artículo de Do et al. (5), el *stake power* (véase figura 3-3). Como en la propuesta de este TFG no se dispone de un contrato de stake (aunque sería posible extender este sistema para incluirlo), se ha decidido utilizar el balance total de la cuenta que corresponde a la dirección del nodo en el sistema de *blockchain*. De esta forma, cuanto más dinero tenga un nodo en su cartera, mayor confianza estará depositando en la red, por lo tanto, se considerará más fiable y se le otorgará mayor reputación.

El segundo factor **N** está basado en el tercer factor también del artículo de Do et al. (5), el *transaction-based reputation rating* (véase figura 3-3), en este caso sin modificación de concepto. Es la cantidad de transacciones realizadas por el nodo, denominado *nonce* en la comunidad de desarrolladores de *blockchain*, porque se valora que un nodo sea activo para que se considere más fiable, es decir, que tenga mayor reputación.

El tercer factor **P** se basa en el tercero del artículo de Zhuang et al. (6), que se describía como "la contribución al consenso", y se ha considerado que la mejor forma de contribuir al consenso es haber producido bloques (si ha sido elegido validador), ya

que, si ya ha validado bloques anteriormente, es más probable que lo pueda volver a hacer de forma fiable, y por ello se le otorga una mayor reputación.

Los pesos (*weights*) $\mathbf{c_i}$ de cada factor tienen los siguientes valores por defecto en el *Smart Contract*:

$$\mathbf{c_1 = 1,\ c_2 = 3}\ \text{y}\ \mathbf{c_3 = 2}$$

Estos pesos son modificables *a posteriori* por el dueño del contrato (el nodo que desplegó inicialmente el contrato, conocido como el *owner*) para realizar ajustes sobre el sistema de reputación, y se pueden consultar por cualquier persona que llame a los métodos de consulta sin coste de gas (por ser métodos tipo *view*). En este diseño se ha decidido utilizar valores enteros para los pesos, pero se podrían modificar fácilmente para que se utilicen factores decimales de coma flotante, como es práctica común en otros sistemas basados en contratos como el estándar de tokens ERC-20 o la propia criptomoneda Ether.

También se planteó usar un factor de aleatoriedad, pero finalmente se descartó por la dificultad de manejar números aleatorios en Solidity, aunque se considera como posible trabajo futuro.

El balance (**B**) se recupera directamente en Solidity, puesto que la clase *address* tiene un atributo para consultarlo.

Sin embargo, el *nonce* (**N**) ha sido más complicado de recuperar, puesto que no se puede conseguir directamente desde Solidity. La solución que se ha alcanzado ha sido enviarlo desde Java cada vez que un nodo envía su voto, y almacenarlo en una estructura de datos *mapping* (pares clave-valor) en el *Smart Contract*. Por seguridad, el *nonce* que se envía con el voto pasa por una validación: no puede ser menor o igual que el anterior que se hubiese enviado (en caso de no haberse enviado antes, debe ser mayor que cero).

En cuanto al número de bloques producidos (**P**), se ha recurrido a una solución directamente en Solidity (esto agrega seguridad, ya que la información recibida desde los clientes no es fiable, pues el código del cliente de Ethereum puede ser manipulado por un atacante). Cada vez que se valida un bloque y se pasa a la siguiente ronda, mediante una llamada al método *nextTurn()* o *nextTurnAndVote()* (que serán descritos

en la sección 6.3.3) se incrementa el número de bloques producidos del nodo que acaba de validar el bloque, manteniendo esta información también en una estructura de datos *mapping* en el contrato.

5.4 Mecanismo de consenso basado en reputación

Después de haber descrito en las secciones anteriores el esquema general del prototipo, su arquitectura y el esquema de reputación que se ha diseñado, en esta sección se describirá el procedimiento general que sigue el mecanismo de consenso basado en reputación, que está representado gráficamente en la figura 5-3.

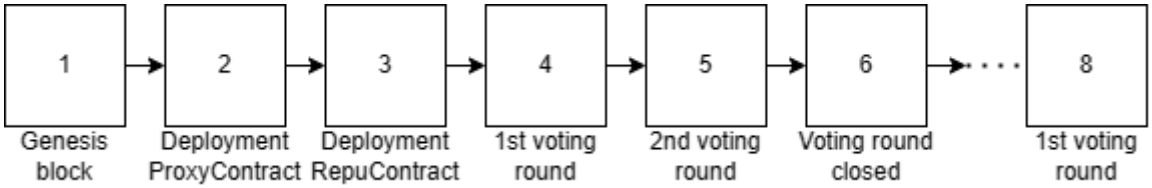

Figura 5-3 Procedimiento general del mecanismo de consenso basado en reputación

En primer lugar, previamente a la creación de la cadena de bloques con mecanismo de consenso Repu, el cliente Hyperledger Besu que va a producir el primer bloque (bloque génesis) debe utilizar un fichero de configuración con las características de este primer bloque. Este fichero, como se describió en detalle anteriormente, contiene parámetros de configuración como el tiempo entre bloques, o una lista de direcciones con un balance (cantidad de dinero) inicial reservado. Para crear una cadena de bloques con el mecanismo de consenso Repu, se debe indicar en los parámetros que se va a utilizar el mecanismo basado en reputación y los parámetros de configuración de este mecanismo, que son los mismos que los de Clique (véase línea 4 de la figura 4-3): el tiempo mínimo para producir el siguiente bloque y el número de bloques hasta la siguiente reiniciación de votos. Este último parámetro permanece para mantener la implementación basada en Clique, pero es reemplazado por el sistema de votación propio del mecanismo Repu que se detalla en la sección 6.3.

El inicio de la red de *blockchain* con este mecanismo de consenso requiere que los primeros bloques se produzcan de una forma particular (el proceso de arranque de la red se explicará con más detalle en la sección 6.4). Primero se debe iniciar un solo nodo en la red, que es el creador de la cadena de bloques. Este nodo creará el bloque

génesis y los dos siguientes, necesarios para desplegar los contratos ProxyContract y RepuContract. Este paso es imprescindible, pues no es posible que el mecanismo de consenso tome el control hasta que estos dos contratos estén desplegados y disponibles para su ejecución.

A partir del bloque número 4, incluido, es el *Smart Contract* RepuContract quien decide quién es el siguiente validador de bloques. Sin embargo, en principio el único validador registrado en el contrato es el creador de la *blockchain* (se podrían fijar más direcciones iniciales en el contrato), y, por lo tanto, hasta la primera ronda de votación seguirá siendo él quien valide los bloques.

Las rondas de votación se realizan en periodos de duración constante, cada N bloques. Cuando el creador de la cadena de bloques despliega los contratos inteligentes que gestionan el mecanismo de consenso, envía como parámetro el valor inicial de la duración de cada periodo de votación, y no es modificable posteriormente en el contrato. Sin embargo, se podría modificar desplegando un nuevo contrato y actualizando los clientes a una nueva versión.

Para que un cliente Hyperledger Besu pueda emitir su voto, este debe ser configurado. Para ello, el cliente debe tener en su directorio local un fichero con el nombre "validatorVote" que tenga como contenido únicamente el *address* del nodo de la red al que quiere votar. El mantenimiento o modificación de este fichero es responsabilidad del administrador del nodo y no es parte de este trabajo.

Para este prototipo se ha utilizado el valor 5. Por lo tanto, cada 5 bloques es turno de votar (en el bloque 5, el 10, el 15, etc.). Sin embargo, el proceso de votación no dura únicamente un bloque, sino que empieza dos bloques antes. En el momento posterior a la producción del bloque dos veces anterior al bloque de votación (por ejemplo, el bloque 3), todos los clientes de la *blockchain* que tengan configurado su voto lo emiten. Un bloque después, tras la producción del bloque inmediatamente anterior al bloque de votación (por ejemplo, el bloque 4), el validador del bloque dos veces anterior emite su voto (por ejemplo, el validador del bloque 3). Finalmente, tras la producción del propio bloque de votación el *Smart Contract* hace el recuento de los votos y cierra la votación, proceso que se detallará a continuación.

Tabla 1 - Esquema de votación de una prueba con 3 nodos

Nodos	1	2	3
Nodo votado	3	3	2
Reputación	56	501	12

Tabla 2 - Resultados de la votación en una prueba con 3 nodos

Nodos	1	2	3
Recuento votos	0	1	2
Votos ponderados	0	12	557
Seleccionados (max. 5)		2	1

Para hacer el recuento de los votos, el contrato los pondera por la reputación del nodo que lo ha emitido. Por ejemplo, en la tabla 1 se puede observar que el nodo 3 emite su voto hacia el nodo 2 (en la segunda fila), y que su reputación es 12 (en la tercera fila); y en la tabla 2 se puede ver que el nodo 2 ha recibido únicamente un voto, del nodo 3 (en la segunda fila), y el valor de ese voto ponderado es 12 (en la tercera fila), ya que la reputación del nodo que lo emitió es también 12.

A continuación, el *Smart Contract* selecciona a los nuevos validadores a partir de los resultados obtenidos. Para ello, se ordena una lista de candidatos a validadores por sus votos ponderados recibidos, y se añaden a la lista de validadores el número máximo de validadores (como se puede observar en la cuarta fila de la Tabla 2), que es un número modificable por el *owner* del contrato, establecido en este prototipo a 5. Si el número de nodos candidatos es mayor o igual que el número máximo, que es el tamaño de la lista de validadores, se eliminarán todos los validadores anteriores para añadir los

nuevos. En caso contrario, si el número de candidatos es menor que el número máximo, se añadirán estos a la lista manteniendo los de mayor reputación que hubiese anteriormente, hasta llegar al número máximo. Por ejemplo, si había 2 validadores y hay 4 candidatos, se eliminará el segundo validador antiguo y se añadirán los 4 candidatos, manteniendo el validador anterior de mayor reputación.

Existen una serie de restricciones por las cuales un candidato a validador puede ser rechazado de ser añadido a la lista. Estas son: que su *nonce* sea 0, es decir, que nunca haya votado y enviado su nonce al *Smart Contract*; que su balance (cantidad de dinero) sea igual a 0, ya que no podría afrontar el consumo de gas para producir bloques; y que su *address* esté en la lista negra de validadores (se describirá en la sección 7.2 las causas por las que un *address* puede entrar en esa lista).

Por último, el *Smart Contract* ordena la lista de nuevos validadores por sus valores de reputación, de forma que el siguiente bloque será validado por el validador con mayor reputación, el siguiente por el validador en la segunda posición de la lista, y así sucesivamente.

A partir del siguiente bloque al bloque de votación (por ejemplo, el 6), no se podrá volver a emitir un voto hasta el comienzo de la siguiente ronda de votación, en la que se repite el mismo proceso. En ese bloque también se hará efectivo el cierre de votación, y, por lo tanto, el siguiente bloque (por ejemplo, el 7) ya será validado por un validador de la nueva lista.

Capítulo 6 - Implementación

En este capítulo se describirán la infraestructura y herramientas que se han utilizado para la implementación del diseño descrito en el anterior capítulo; los componentes más relevantes que se han desarrollado en la implementación del diseño, que son el mecanismo de consenso basado en un contrato inteligente, y el cliente Hyperledger Besu; y también se incluirá una guía de instalación del prototipo.

6.1 Infraestructura y herramientas utilizadas

La infraestructura utilizada para la implementación de este trabajo ha sido un ordenador con el sistema operativo Windows y el subsistema Linux para Windows (WSL). Las herramientas utilizadas han sido el IDE IntelliJ IDEA para Java, el IDE Remix para Solidity y pruebas con los *Smart Contracts*, Visual Studio Code y Notepad++ para editar diferentes archivos.

En WSL se ha utilizado el editor de archivos "gedit" así como el compilador solc de Solidity para compilar los contratos y obtener el código binario compilado, que debía situarse en el código Java, en las clases que interactuaban con ellos. Para instalar el compilador se siguió la documentación Oficial de Solidity (17).

Una vez instalado, el comando utilizado para compilar es el siguiente:

```
solc --bin RepuContract.sol
```

Se ha utilizado la librería Web3j (16) en Java, que es un cliente web3 para comunicarse con los *Smart Contracts* de una *blockchain*.

Para el control de versiones se ha utilizado GitHub, conectado al IDE IntelliJ IDEA.

Para ejecutar el cliente Hyperleger Besu se han utilizado terminales de Windows, primero para compilar el proyecto y después una por cada nodo en la cadena de bloques.

Para utilizar la API de Hyperledger Besu existían dos posibilidades, utilizar el comando "curl" en la terminal de Windows, o un cliente Postman. Se ha optado por utilizar el cliente Postman por tener una interfaz intuitiva.

6.2 Componentes más relevantes

En esta sección se va a describir la implementación del diseño de los componentes más relevantes del prototipo. Estos son el mecanismo de consenso basado en un contrato inteligente y el cliente Hyperledger Besu,

6.2.1 Implementación del mecanismo de consenso basado en *un Smart Contract*

Partiendo del diseño descrito en la sección 5, se han implementado dos *Smart Contracts*. El primero es **ProxyContract**, que es un contrato sencillo que contiene únicamente la dirección del contrato de reputación y dos métodos para recuperar la dirección y poder actualizarla, de forma que el contrato de reputación sea modificable una vez que la *blockchain* ya haya sido creada. El segundo es **RepuContract**, que es el contrato que contiene la mayor parte de la lógica del mecanismo de consenso basado en reputación, y con el que se comunican los clientes de la cadena de bloques. A continuación, se describirá el proceso para la implementación de estos contratos inteligentes.

Esta idea de utilizar un contrato que implemente parte del mecanismo de consenso que es invocado directamente desde el cliente Hyperledger Besu es muy innovadora, por lo que al inicio del proyecto se desconocía si existían limitaciones técnicas que impidieran su implementación. Por este motivo, antes de implementar completamente el mecanismo de consenso, se realizaron diversas pruebas para ir resolviendo de forma incremental los problemas técnicos derivados de este enfoque. A continuación, se describen las pruebas realizadas.

La primera prueba consistió en programar un contrato simple, un contador (ver figura 6-1), con métodos sencillos: incrementar, recuperar, y establecer una variable de tipo entero; con el fin de aprender a utilizar Web3j, el cliente web3 de Java, y probar la integración con Hyperledger Besu. Para ello se hicieron algunas modificaciones en el mecanismo de consenso Clique añadiendo una clase Java que realizara la comunicación con el contrato, y algunos mensajes de log durante la producción de bloques para comprobar el funcionamiento de este, mostrando el resultado de las llamadas a los métodos.

```
pragma solidity >=0.7.0 <0.9.0;

//Simple counter

contract Contador{
    uint256 count = 1;
    function setCount(uint256 _count) public {
        count=_count;
    }
    function incrementCount() public {
        count+=1;
    }
    function getCount() public view returns(uint256) {
        return count;
    }
    function getNumber() public pure returns(uint256) {
        return 34;
    }
}
```

Figura 6-1 Smart Contract contador, utilizado como prototipo de contrato para Web3j

Una vez conseguido que este prototipo de contrato sencillo funcionase correctamente en Hyperledger Besu, el siguiente paso fue separar estos cambios de Clique y dejarlo en su estado original. Para esto se creó una nueva carpeta dentro del directorio consensus llamada "repu" (así se ha llamado al mecanismo de consenso basado en reputación que se ha implementado), y se hizo un clonado completo de Clique, tanto de su carpeta individual, como del resto de referencias que había en todo el proyecto de Besu.

Después de esto, se empezó a crear un contrato de consenso prototipo, que solo contuviese métodos simbólicos como la obtención del siguiente validador, actualización manual del siguiente validador, entre otros, con el fin de tener un prototipo al que llamar desde Besu, y modificar el funcionamiento de este para que tuviese en cuenta la información recibida desde el contrato, en lugar de la que usaba Clique. Por ejemplo, si un nodo es validador o no, quién es el siguiente validador, y que los nodos se sincronizasen entre ellos.

Una vez Hyperledger Besu estuvo funcionando correctamente con un contrato prototipo, el siguiente paso fue implementar la lógica y complejidad en Solidity del contrato de reputación completo, que se describirá a continuación.

6.2.1.1 Contrato RepuContract

En la figura 6-2 se muestran los componentes principales del contrato RepuContract que implementa el sistema de reputación: estructuras de datos, funciones públicas y modificadores. Los modificadores son componentes característicos del lenguaje Solidity de programación de contratos, que permiten establecer restricciones de seguridad sobre la ejecución de funciones públicas. Los modificadores se describen en detalle en la sección 7.2 de seguridad del sistema. A continuación, se describen las estructuras de datos y las funciones públicas del contrato RepuContract.

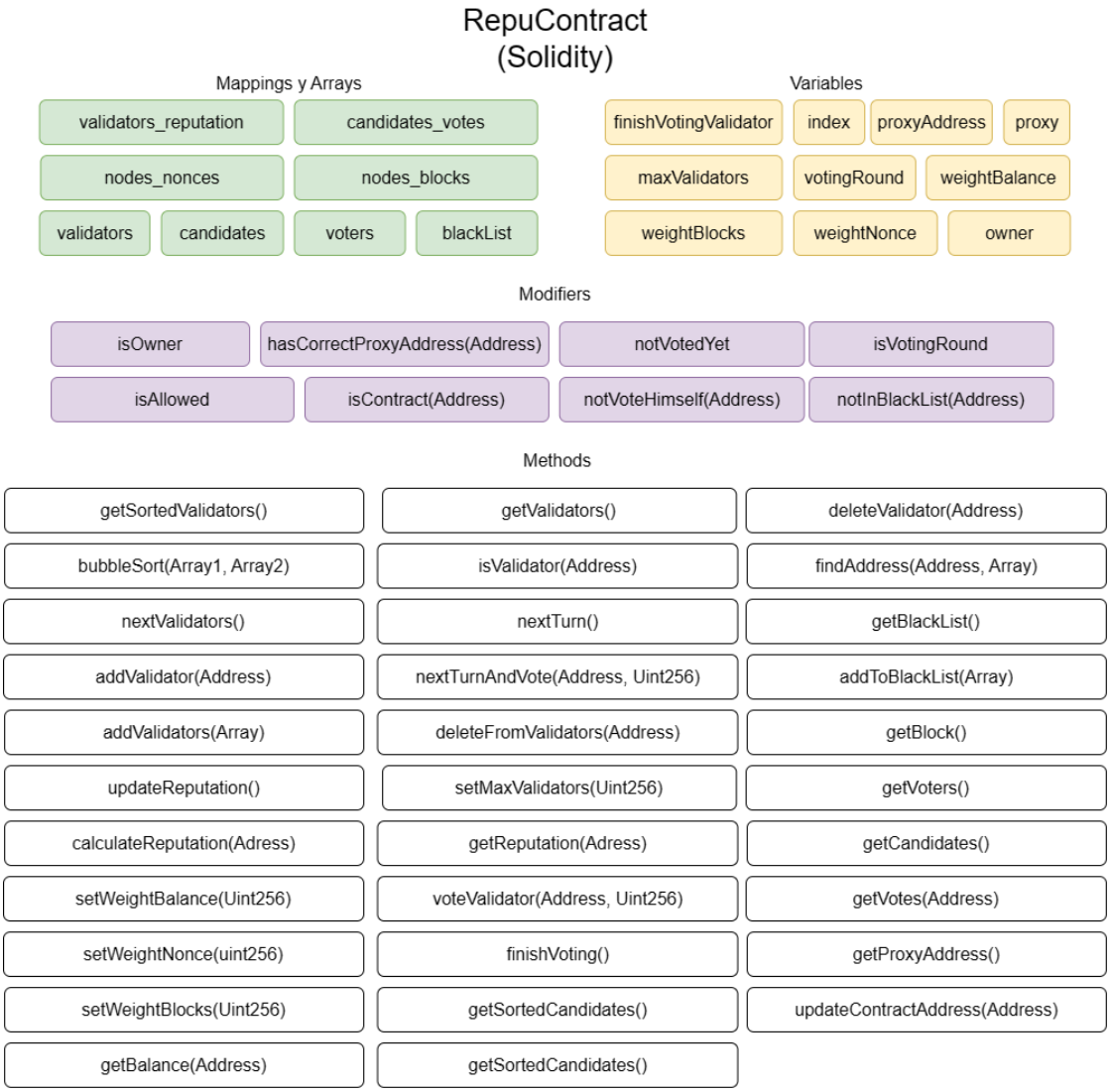

Figura 6-2 Diagrama que representa las estructuras de datos, modificadores y funciones públicas del Smart Contract de Repu para gestionar el mecanismo de consenso

Estructuras de datos

validators_reputation es un *mapping* (estructura de datos similar a una tabla hash que almacena pares clave-valor) que contiene la reputación de cada validador.

candidates_votes es un *mapping* que contiene los votos de cada candidato a validador. Se reinicia al final de cada ronda de votación.

nodes_nonces es un *mapping* que contiene el *nonce* de cada nodo que haya votado alguna vez.

nodes_blocks es un *mapping* que contiene el número de bloques validados por cada nodo.

validators es un array que contiene los *address* de los nodos validadores en cada momento.

candidates es un *array* que contiene los candidatos a ser validador en la ronda de votación. Se reinicia al final de cada una.

voters es un *array* que contiene la lista de votantes en cada ronda de votación. Se utiliza para validar que cada nodo no vote más de una vez. Se reinicia al final de cada una.

blackList es un *array* que contiene la lista de nodos en la lista negra, que no podrán votar ni ser validadores nunca más.

finishVotingValidator es una variable que contiene al validador que ha ejecutado el método *finishVoting*(). Se almacena para el caso especial en que el validador deja de serlo en esa ronda de votación, y debe ejecutar el método *nextTurn*() pasando por una validación que comprueba si el nodo que lo ejecuta es validador.

index es el índice del *array* de validadores (módulo tamaño del *array*) en el que se encuentra la red. Se reinicia a 0 al final de cada ronda de votación.

proxyAddress y ***proxy*** son el *address* y la instancia del contrato Proxy, respectivamente.

maxValidators es el número máximo de validadores que puede haber al mismo tiempo. Es modificable mediante su método setter.

votingRound es el periodo de votación, es decir, cada cuántos bloques es momento de votar. Es modificable mediante su método setter.

weightBalance, ***weightBlocks***, y ***weightNonce*** son los pesos de cada factor de reputación. Son modificables mediante sus métodos setter.

owner es el *address* del creador del contrato, es decir, el nodo que lo desplegó. Se utiliza para validar quien ejecutar algunos métodos restringidos al creador del contrato.

Métodos (funciones públicas)

getValidators(), ***getBlackList***(), ***getVoters***(), ***getCandidates***(), y ***getProxyAddress***() son métodos *getter* estándar que devuelven el array o variable correspondiente.

deleteValidator() y ***deleteFromValidators***() son dos métodos que eliminan el *address* indicado de la lista de validadores. El segundo también llama a *updateReputation*() y solo el *owner* puede ejecutarlo.

bubleSort() es un método que implementar el algoritmo de ordenación iterativo y ordena dos *arrays* (de igual tamaño) por orden de mayor a menor según los valores del primer *array*.

isValidator() es un método que comprueba si el *address* indicado es un nodo validador.

findAddress() es un método que busca en el *array* el *address* indicado, devuelve la posición en la que está si lo ha encontrado, y la última posición más uno en caso de no encontrarlo.

nextValidators() es un método que devuelve la lista de validadores actuales empezando por la posición de la variable *index* módulo el tamaño del *array*.

nextTurn() y ***nextTurnAndVote***() son dos métodos que incrementan en uno el índice de validadores *index*, incrementan en uno los bloques producidos por el validador actual, y, si se encuentran en el bloque siguiente al turno de votación, ejecutan *finishVoting*() y reinician el *index*. El segundo método, si se encuentra en turno de votación, ejecuta *voteValidator()*.

addValidator() y ***addToBlackList***() son dos métodos que añaden un *address* indicado a la lista de validadores y a la lista negra, respectivamente.

addValidators() es un método que añade una lista de validadores a la lista de validadores hasta llegar al límite máximo. Lleva a cabo una serie de validaciones en cada *address* de la lista, que son si su *nonce* es mayor que 0, si no está en la *black list* y si su balance es mayor que 0.

getBlock() es un método que devuelve el bloque actual.

updateReputation() es un método que actualiza la reputación de cada validador en el mapping *validators_reputation* llamando a *calculateReputation()*. Después, ordena la lista de validadores llamando a *getSortedValidators()*.

setMaxValidators(), ***setWeightBalance***(), ***setWeightNonce***(), y ***setWeightBlocks***() son métodos setter estándar que actualizan el valor de la variable correspondiente.

calculateReputation() es un método que calcula y devuelve la reputación de un *address* indicado en base a los tres factores y los tres pesos.

getReputation() y ***getVotes***() son dos métodos que devuelven la reputación y los votos recibidos (como candidato) de un *address* indicado, respectivamente. En el caso del primero, hace una llamada al método *calculateReputation*(), el segundo directamente lee del *mapping candidates_votes*;

voteValidator() es un método que realiza la votación de un nodo votante hacia el *address* indicado, tras llevar a cabo una serie de validaciones, que son que se encuentre en periodo de votación, que aún no haya votado, que no se vote a sí mismo, y que ni el nodo al que está votando ni él mismo se encuentren en la lista negra. También recibe el *nonce* actual del votante, y comprueba que sea mayor estricto que el anterior registrado. Además, ordena la lista de candidatos por número de votos llamando a *getSortedCandidates()*.

finishVoting() es un método que cierra la ronda de votación en el siguiente bloque al turno de votar. Reinicia las listas de *voters, candidates* y el mapping *candidates_votes*, añade los validadores con *addValidators()* y actualiza las reputaciones con *updateReputation()*.

getSortedValidators() y ***getSortedCandidates***() son dos métodos que llaman a *bubbleSort()* para ordenar la lista de validadores por reputación y la de candidatos por votos, respectivamente, y las devuelven.

updateContractAddress() es un método, que solo puede ejecutar el *owner*, que actualiza el address del contrato (de sí mismo) llamando al método externo *setConsensusAddress*() de la instancia del contrato Proxy. Realiza dos validaciones, que son que el nuevo contrato (con el *address* indicado) contenga el mismo *address* del contrato Proxy, y que el *address* indicado pertenezca efectivamente a un *Smart Contract*.

getBalance() y ***getProducedBlocks***() son dos métodos que devuelven los factores de reputación de balance y número de bloques producidos, respectivamente.

6.2.1.2 Contrato ProxyContract

Este contrato sirve de *proxy* del contrato RepuContract, de forma que permite obtener y modificar la dirección del contrato que implementa el sistema de votación. En la figura 6-3 se muestran los distintos componentes que forman este contrato y que se describen a continuación:

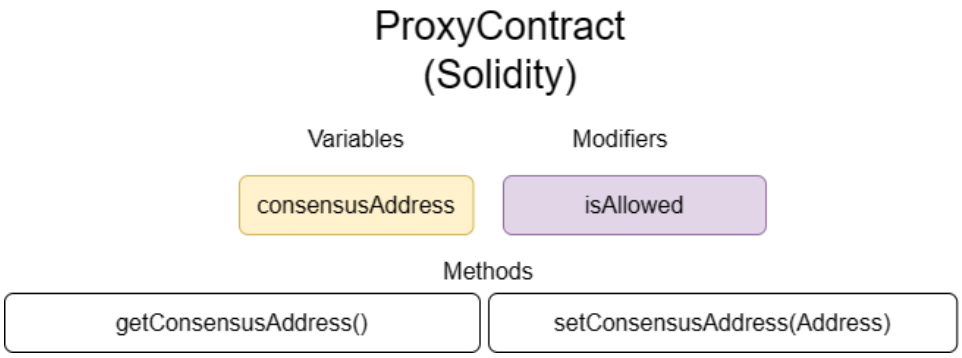

Figura 6-3 Diagrama que representa las variables, modifiers y métodos del Smart Contract Proxy

consensusAddress es una variable que contiene el *address* del contrato de reputación.

isAllowed es un *modifier* que comprueba que el emisor de la transacción es igual al actual *consensusAddress*.

getConsensusAddress() y ***setConsensusAddress***() son métodos *setter* y *getter* estándar de la variable *consensusAddress.* En el caso del setter, utiliza el *modifier isAllowed* para restringir que solamente el contrato RepuContract actual pueda ejecutarlo.

6.2.2 Implementación del nuevo mecanismo de consenso en el cliente *Hyperledger Besu*

Como se ha mencionado anteriormente, Hyperledger Besu (8) es un cliente Ethereum escrito en Java, que permite crear cadenas de bloques para redes privadas, y se ha seleccionado como base para la implementación del prototipo de mecanismo de consenso basado en reputación. Partiendo del diseño descrito en la sección 5, se describirá en esta sección la implementación que se ha llevado a cabo.

En el siguiente enlace se puede acceder al repositorio GitHub que se ha utilizado para el control de versiones, a una comparación entre la rama master (un clone del proyecto original de Besu) y la rama "dev" en la que se han subido los *commits* y realizados los cambios:

https://github.com/Danniilpz/besuPoR/commit/d2b65de17ba502c2cf478ed918e1c523f5586ecd

Se ha utilizado dos clases que implementan la comunicación entre el cliente Hyperledger Besu y los *Smart Contracts* con la librería *Web3j* (16), **ProxyContract** y **RepuContract**. La primera se encarga de interactuar con el *Smart Contract* ProxyContract, y la segunda con RepuContract. Ambas contienen un método Java por cada función de Solidity, haciendo la conversión de tipos nativos de Solidity (soportados en la librería Web3j) a tipos de Java.

Por otra parte, la mayor parte de la funcionalidad añadida sobre el código original de Hyperledge Besu se ha implementado en la clase **RepuHelpers**, que contiene los métodos llamados desde la clase de producción de bloques **RepuBlockMiner**, que también ha sido modificada respecto a la original.

6.2.2.1 Clase RepuBlockMiner

Como se acaba de indicar, la clase RepuBlockMiner se encarga de producir el siguiente bloque en la cadena de bloques cuando el nodo que ejecuta el cliente

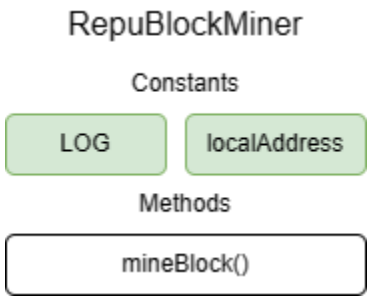

Figura 6-4 Diagrama que representa las constantes y métodos de la clase RepuBlockMiner en el nodo Besu

Hyperledger Besu es el elegido por el sistema de reputación. En la figura 6-4 se muestra el contenido de esta clase.

En esta clase se ha modificado el **constructor** para obtener algunos atributos que eran necesarias para las clases que utilizan *Web3j*. Estos son: el *NodeKey*, que contiene la clave privada y la clave pública del nodo actual, y es necesario para obtener las credenciales para poder firmar transacciones; y el puerto en el que se ha abierto el nodo, que es necesario para inicializar la instancia de *Web3J*. También se llama a un método de RepuHelpers *getRepuContract()*, cuya funcionalidad es instanciar el contrato cuando ya haya sido desplegado por el nodo creador de la *blockchain* (a partir del bloque número 3).

Una dificultad adicional que presenta la implementación del sistema de reputación en un contrato es la forma de comunicar a los distintos nodos la dirección del contrato ProxyContract una vez se ha desplegado en la cadena de bloques. Como el *address* de un contrato desplegado por un cierto nodo con una cierta clave privada siempre es el mismo, se ha decidido fijar el *address* del contrato Proxy en una constante en el código fuente de Hyperledger Besu, y el *address* del contrato de consenso inicialmente está fijado en una variable, pero en cada bloque se comprueba en el proxy si el *address* ha cambiado, por medio de llamadas al método *getConsensusAddress*() de ProxyContract para, en ese caso, actualizarlo. De esta forma, se resuelve el problema

inicial de cómo informar al resto de nodos del *address*, pues siempre es el mismo en el caso del Proxy, y el del contrato de consenso se puede consultar en este.

Además del constructor, en RepuBlockMiner se ha modificado el método ***mineBlock****()*, que es el que se ejecuta para minar un bloque. En el diagrama de la figura 6-5 se muestra el funcionamiento de este método.

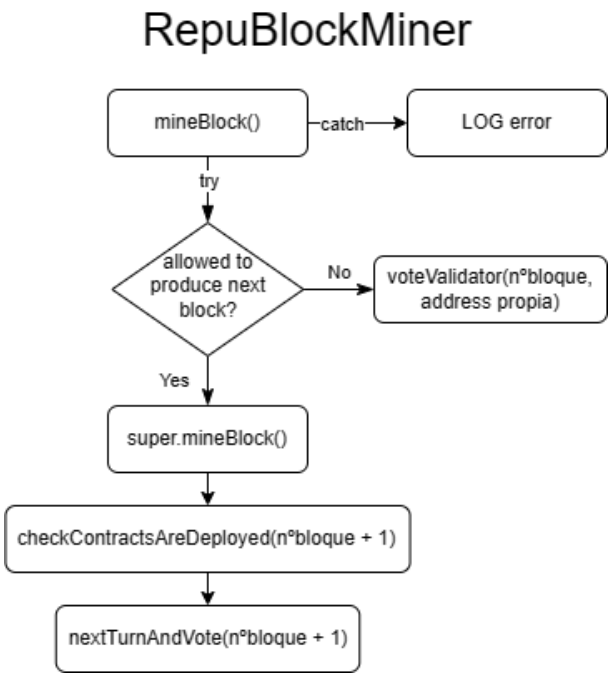

Figura 6-5 Funcionamiento del método mineBlock() de RepuBlockMiner en Hyperledger Besu

Los métodos ***addressIsAllowedToProduceNextBlock****()*, ***voteValidator****()*, ***checkContractsAreDeployed****()* y ***nextTurnAndVote****()* son métodos *static* de la clase RepuHelpers.

El método ***addressIsAllowedToProduceNextBlock****()* que se llama en la condición comprueba si el nodo actual debe producir bloque en esta ronda o no.

El método ***checkContractsAreDeployed****()* se encarga de desplegar los contratos Proxy y RepuContract si no están desplegados.

El método ***nextTurnAndVote****()* se encarga de pasar al siguiente turno de validador y, si está en la ronda de votación, de enviar el voto del nodo que está produciendo

bloque (el cambio de turno y el voto deben ir en la misma transacción, pues no se pueden enviar dos transacciones del mismo nodo en el mismo bloque, porque el hilo que lanza la transacción se queda en espera).

El método ***voteValidator()*** es llamado por los nodos que no producen bloque en esta ronda, y se encarga de enviar el voto en caso de estar en ronda de votación. En caso contrario, no hace nada.

6.2.2.2 Clase RepuHelpers

La clase RepuHelpers es la que implementa la mayor parte de la funcionalidad del mecanismo de consenso en el cliente Hyperledger Besu. Contiene los métodos que hacen de intermediarios entre la clase productora de bloques (RepuBlockMiner, que se ha descrito en la sección anterior), y los *Smart Contracts*.

A continuación, se describen los componentes de la clase RepuHelpers, representada gráficamente en la figura 6-6.

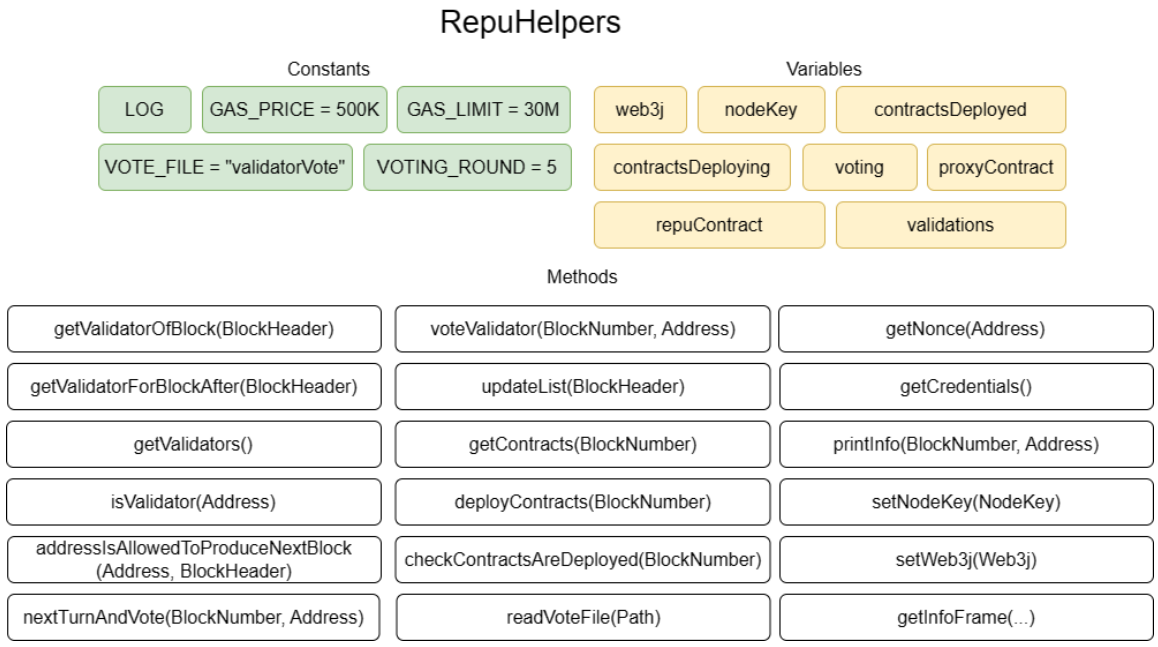

Figura 6-6 Diagrama que representa las constantes, variables y métodos de la clase RepuHelpers en el nodo Besu

LOG es una constante utilizada para imprimir mensajes *log* en la terminal.

GAS_PRICE y ***GAS_LIMIT*** son constantes que indica el precios y límite de gas.

VOTE_FILE es una constante que indica el nombre del fichero local en el que el nodo configura su voto para ser enviado.

VOTING_ROUND es una constante que indica cada cuántos bloques producidos debe haber una ronda de votación.

web3j, ***nodeKey***, ***proxyContract*** y ***repuContract*** son variables que contienen instancias de sus respectivas clases.

contractDeployed y ***contractDeploying*** son variables booleanas que indican si los contratos ya han sido desplegados y si los contratos están siendo desplegados en este momento, respectivamente. Se utilizan para la sincronización entre hilos del nodo que se está ejecutando, evitando que dos hilos ejecuten el mismo método, que solo debe ejecutarse una vez.

voting es una variable que indica si el nodo está actualmente votando. Se utiliza también para la sincronización entre los hilos del propio nodo.

validations es una variable que contiene una estructura *map* con parejas clave-valor que contienen número de bloque y validador. Se utiliza para la sincronización de los hilos de un nodo, así como para la sincronización de los diferentes nodos, ya que cada validador se obtiene a partir del *Smart Contract*.

getValidatorOfBlock(), ***getValidatorForBlockAfter***(), ***getValidators***() y ***isValidator***() son métodos que comprueban el validador de un bloque, el siguiente validador, la lista de validadores, y si el nodo con el *address* indicado es validador, respectivamente. Para ello se usan las llamadas a los métodos *nextValidators*() y *isValidator*() del *Smart Contract*.

addressIsAllowedToProduceNextBlock() es un método que comprueba si el nodo actual es un validador y si es su turno de producir bloque.

nextTurnAndVote() y ***voteValidator***() son métodos que, en caso ser turno de votación, y de que el nodo tenga configurado su voto (tenga en su carpeta local el fichero "validatorVote" no vacío y con el formato correcto), envían el voto. Además, el primer método es llamado por los validadores en el momento inmediatamente posterior a

producir bloque, y en caso de no estar en turno de votación, llama al método *nextTurn*() del *Smart Contract*, para dar paso al siguiente validador.

updateList() es un método que actualiza la estructura map "validations" con el bloque indicado y el siguiente validador, que se obtiene con una llamada al método *nextValidators*() del *Smart Contract* y obteniendo el primer elemento de la lista.

getContracts(), ***deployContracts***(), y ***checkContractsAreDeployed***() son métodos relacionados con los *Smart Contracts*. Los dos primeros se encargan de instanciar las clases ProxyContract y RepuContract. El primero se utiliza cuando la *blockchain* ya está creada y ha superado los bloques génesis (bloque mayor o igual que 3). El segundo es utilizado por el nodo creador de la *blockchain* para desplegar los contratos en los bloques génesis. El tercer método simplemente comprueba si se deben desplegar los contratos y, si es así, llamar al método de despliegue.

readVoteFile() es un método que se encarga de leer el fichero de voto del nodo, en caso de existir y tener el formato correcto.

getNonce() es un método que devuelve el *nonce* más uno (número de transacciones) de un *address* dado a partir de la instancia de *Web3j*.

getCredentials() es un método que devuelve una instancia de la clase *Credentials* a partir de las claves pública y privada obtenidas con el objeto *NodeKey*.

printInfo() es un método que imprime en la terminal mensajes de *log* mostrando información sobre el mecanismo de reputación. Esta información es el número de votos tras una ronda de votación, y, si un nodo es elegido validador, su reputación.

setNodeKey() y ***setWeb3j***() son métodos *setters* estándar para las instancias de *NodeKey* y Web3j.

getInfoFrame() es un método que devuelve un mensaje log en formato *frame* con información sobre el sistema de votación en sus diferentes fases, para ser visualizado en las terminales durante la ejecución (ver figura 6-7, figura 6-8 y figura 6-9)

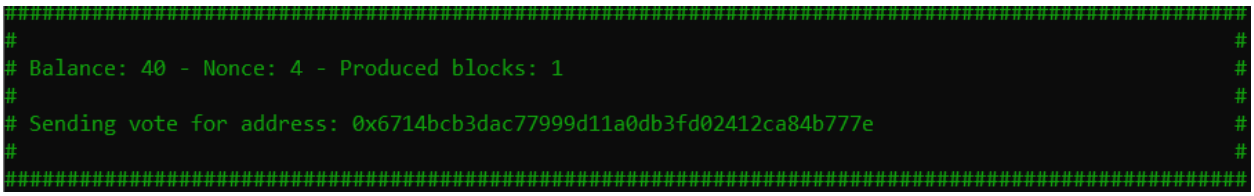

Figura 6-8 Frame generado por Besu en la fase intermedia de votación

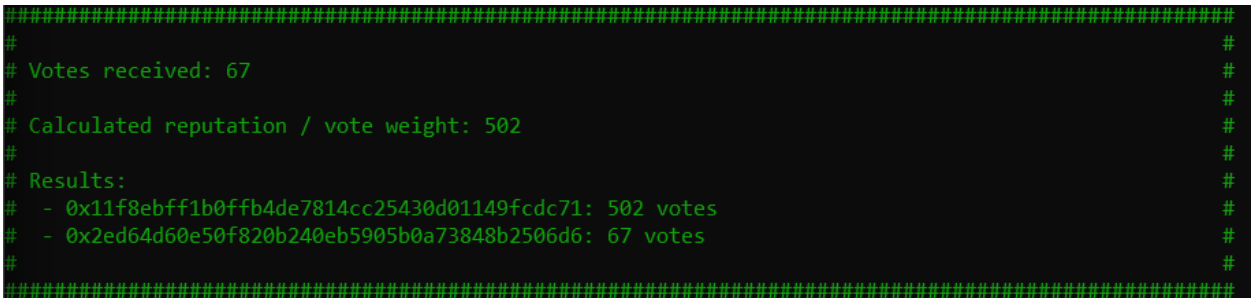

Figura 6-7 Frame generado por Besu en la primera fase de votación

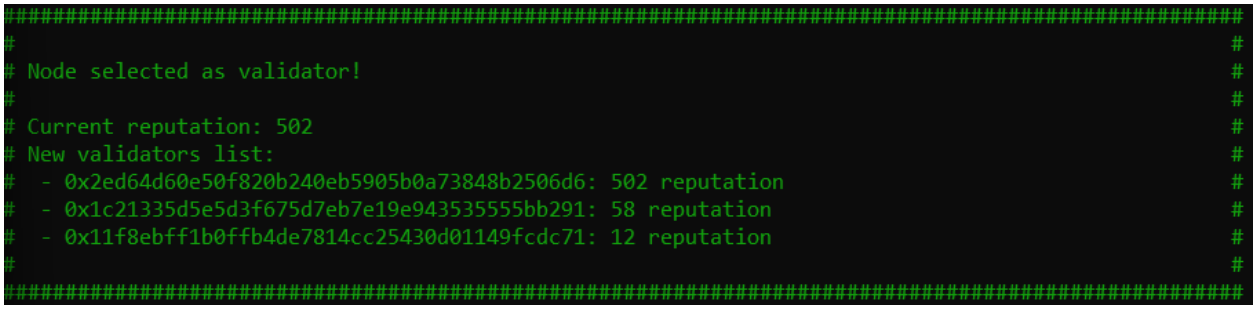

Figura 6-9 Frame generado por Besu en la fase de cierre de votación

6.3 Implementación del sistema de votación

Como se ha indicado anteriormente, el mecanismo de consenso basado en reputación incluye un sistema de votación para que los nodos validadores puedan elegir nuevos nodos de la red para que sean a su vez validadores. Esto se realiza en rondas de votación que se realizan en periodos de votación de duración constante, fijado en el prototipo en 5 bloques: cada cinco bloques minados (el 5, el 10, el 15, etc) es turno de votar.

El periodo de votación se establece mediante una variable en el cliente Hyperledger Besu, *votingRound*, que se envía al contrato en el constructor cuando se despliega la primera vez. Una vez desplegado el contrato, solo se podría modificar

desplegando un nuevo contrato y modificando el cliente (sería necesario actualizar a una nueva versión).

Cuando el bloque dos veces anterior al bloque de votación es minado, todos los nodos de la red que tengan configurado su voto lo envían en una transacción, a excepción del nodo validador que esté en turno de producir bloque. Debido a esto, se ha alargado un bloque más el periodo de votación, de forma que en el bloque inmediatamente anterior al bloque de votación es minado, los nodos que no pudiesen votar en el primero, enviarán su voto. Después de cada uno de estos dos bloques, se habrá generado una lista de transacciones pendientes que se añaden a la *blockchain* en los bloques siguientes, el bloque módulo de 5 menos 1, y el propio bloque módulo 5.

En el bloque siguiente al bloque de votación, es decir, el bloque módulo de 5 más 1 (el 6, el 11, el 16…), automáticamente se cierra la votación y no se permite votar hasta el siguiente periodo.

Para configurar el voto, cada nodo debe tener un fichero en su carpeta personal "validatorVote" que contenga el *address* del nodo que se quiere votar. El voto de cada nodo se pondera por la reputación de este, que se habrá calculado dinámicamente al enviar el voto con el *nonce*, obteniendo el balance en Solidity y su número de bloques producidos anteriormente; como se describió en apartados anteriores.

Una vez cerrada la votación, se ordena la lista de validadores votados por orden de votos, y se añaden a la lista de nodos validadores, que tiene un límite máximo (modificable por el *owner* del contrato). Si está llena, se van eliminando nodos que no se hayan votado en esta ronda para añadir los nuevos. Una vez añadidos, se ordena la lista por reputación de los validadores, de forma que el primero en validar es el de mayor reputación, y se va rotando la lista en cada bloque producido, es decir, primero será el turno de el que esté en la primera posición de la lista, después el que esté en la segunda, y así sucesivamente hasta el final de la lista, que volverá a empezar por el principio.

Un validador no puede ser elegido si nunca ha votado, es decir, si su *nonce* es cero. Y si en la votación envía un *nonce* menor o igual que el que había enviado anteriormente, se añadirá el validador a la lista negra y nunca podrá votar ni ser validador en el futuro.

A partir del bloque dos veces posterior al bloque de votación (inclusive), empiezan a producir bloques los validadores de la nueva lista creada tras la ronda de votación.

En la implementación que se ha hecho, los 3 primeros bloques de la cadena de bloques se consideran bloques génesis, puesto que, en el primero de los bloques, el bloque cero, se crea la cadena, en el segundo bloque se despliega el contrato Proxy, y en el bloque número 3 se despliega el contrato de reputación. Estos 3 bloques son desplegados por el creador de la cadena de bloques. A partir del bloque 3, el siguiente validador es decidido por el contrato de reputación y no por Besu, el creador puede dejar de ser validador si en la siguiente votación no es votado. Aunque la documentación de Hyperledger Besu [15] indica la posibilidad de desplegar un contrato directamente en el bloque génesis, en la práctica se producían errores que obligaron a este enfoque alternativo utilizando los tres primeros bloques.

El algoritmo de ordenación que se ha utilizado es *Bubble Sort* (18), un algoritmo iterativo clásico. Se intentó utilizar un algoritmo recursivo *QuickSort*, pero no fue posible, ya que Besu revertía las transacciones. Este problema se explicará en detalle en la sección 7.1.

Como ya se ha mencionado anteriormente, durante el desarrollo se encontraron algunos problemas con la sincronización entre los diferentes hilos del nodo y los diferentes nodos entre sí con las funcionalidades de desplegar los contratos, votar y producir bloque.

Para solventar el problema entre los hilos de un mismo nodo en las funcionalidades de desplegar y votar, se han utilizado algunas variables estáticas, que son *contractsDeployed*, *contractsDeploying* y *voting*. Se decidió usar variables porque era necesario que los métodos de despliegue de contratos, así como el de votación, solo se ejecutasen una vez (por un solo hilo).

En cuanto a la sincronización entre nodos, se utilizó el *Smart Contract* como intermediario, con la llamada al método *nextValidators*(), que devuelve el mismo resultado independientemente del nodo que lo llame. En local se almacena en la variable estática *validations* parejas bloque-validador, con el fin de que todos los nodos

se queden a la espera hasta que en ese diccionario aparezca el actual bloque a producir y quien debe producirlo.

6.4 Instalación de Hyperledger Besu

En esta sección se incluirá una guía de instalación del código fuente del cliente Hyperledger Besu modificado con el nuevo mecanismo de consenso "Repu" basado en sistemas de reputación.

Para instalar la versión modificada de Besu con el mecanismo Repu, se debe descargar el código del repositorio GitHub: Danniilpz/besuPoR at dev (github.com)

En el directorio /demo se han incluido los ficheros necesarios para realizar una demo con una red de 4 clientes Besu. Para seguir esta guía se pueden utilizar, o bien crear una red nueva.

Después de haber descargado el repositorio, se debe abrir una terminal y situarse en la carpeta raíz del proyecto y ejecutar el comando "./gradlew installDist", para recompilar el proyecto.

El último paso es configurar la variable de entorno correspondiente para ejecutar Hyperledger Besu. En Windows, se puede buscar "variables de entorno" en el buscador y acceder a la opción del panel de control. (ver figura 6-10)

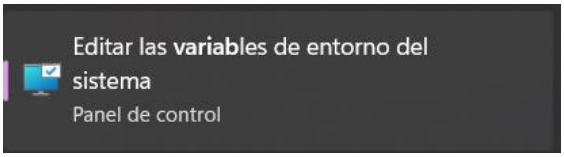

Figura 6-10 Opción de configuración de variables de entorno en Windows 11

Una vez allí, hacer click en "Variables de entorno…". (ver figura 6-11).

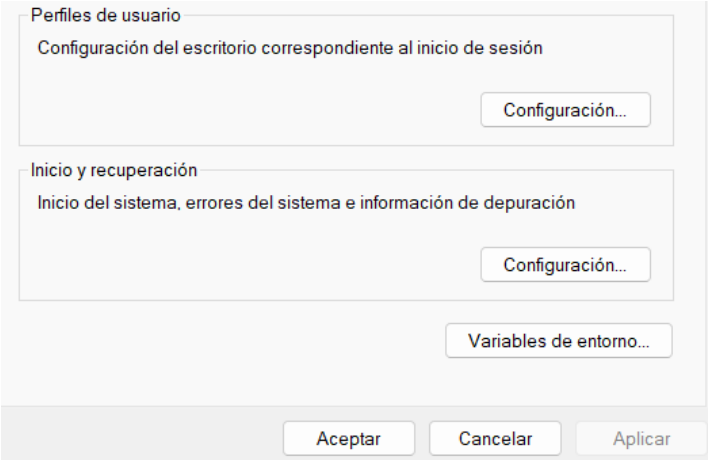

Figura 6-11 Menú de configuración de variables de entorno en Windows 11

Después, hacer doble click en Path de variables de usuario, y añadir la variable "<path>\build\install\besu\bin" siendo "<path> la ruta al directorio raíz del proyecto.

Una vez compilado el proyecto y añadida la variable de entorno, ya se puede montar una pequeña red de nodos Hyperledger Besu de prueba. Para ejecutar una red de nodos en local en Windows, se siguió la documentación de Hyperledger Besu, concretamente las secciones relacionadas con Clique. (19) (20) A continuación, se explicarán los pasos de esta documentación adaptados al nuevo mecanismo de consenso Repu.

En primer lugar, se debe crear una estructura de carpetas, una por cada nodo, con una subcarpeta data dentro. Después obtener el address del nodo *owner* de la blockchain, con el comando:

```
besu --data-path=data public-key export-address --to=data/node1Address
```

Posteriormente, crear el fichero génesis en la carpeta raíz de la estructura de carpetas, con el formato indicado en la documentación. Sin embargo, para adaptar el fichero al nuevo mecanismo de consenso Repu, hay algunos parámetros que deben ser diferentes. (ver figura 6-12). Se puede tomar como ejemplo el fichero "repuGenesis_demo.json" situado en el directorio /demo.

```
"config":{
    "chainId":1337,
    "berlinBlock": 0,
    "repu":{
      "blockperiodseconds":10,
      "epochlength":30000
    }
  }
…
"gasLimit":"0xa00000000000",
```

Figura 6-12 Parámetros de configuración de Repu en el archivo del bloque génesis

Además, se debe añadir en la lista "alloc" una entrada para el *address* del nodo *owner* (que se puede consultar en "Node-1/data/node1Address"), reservando un balance inicial suficiente para poder afrontar el coste de gas que supone desplegar los contratos ProxyContract y RepuContract.

En extraData se debe reemplazar "<Node 1 Address>" por el address del nodo *owner* excluyendo el prefijo 0x.

Por último, para ejecutar el nodo *owner* se debe abrir una termina en su directorio (Node-1), y ejecutar el siguiente comando, modificando "<nombreFicheroGenesis.json>" por el nombre elegido para este.

```
besu --data-path=data --genesis-file=../<nombreFicheroGenesis.json> --network-id 123 --rpc-http-enabled --rpc-http-api=ETH,NET,REPU --host-allowlist="*" --rpc-http-cors-origins="all"
```

Para ejecutar otros nodos se ejecutará el siguiente comando desde la carpeta raíz de cada nodo, sustituyendo <Node-1 Enode URL> por el *enode* del node-1 (se copia en la terminal del Nodo *owner*), así como "<nombreFicheroGenesis.json>". También se deberá cambiar el p2p-port y el –rpc-http-port por cada nuevo nodo que se ejecute, ya que no puede repetirse el puerto.

```
besu --data-path=data --genesis-file=../ <nombreFicheroGenesis.json> --bootnodes=<Node-1 Enode URL> --network-id 123 --p2p-port=30304 --rpc-http-enabled --rpc-http-api=ETH,NET,REPU --host-allowlist="*" --rpc-http-cors-origins="all" --rpc-http-port=8546
```

Opcionalmente, se puede configurar el voto de cada nodo para la ronda de votación de validadores, creando un fichero con nombre "validatorVote" en su directorio, y escribiendo dentro el *address* del nodo al que quiere votar, sin espacios ni antes ni después.

Una vez ejecutados los nodos, ya se puede ver a través de las diferentes terminales el funcionamiento de la cadena de bloques.

Para volver a ejecutar desde el principio la red, se deben primero vaciar las carpetas de los nodos (a excepción de los archivos de *address*, *privateKey* y *validatorVote*). Realizar esta tarea de forma manual es tedioso, por ello se creó un script bash para automatizarla (ver figura 6-1).

```
      if exist ".\Node-1\data\caches" rd /s /q ".\Node-1\data\caches"
      if exist ".\Node-1\data\database" rd /s /q ".\Node-1\data\database"
      if exist ".\Node-1\data\DATABASE_METADATA.json" del /q ".\Node-
1\data\DATABASE_METADATA.json"
```

Figura 6-13 Script bash para la automatización de la tarea de reiniciar la red de nodos local

Este script se debe copiar en un archivo con extensión .bat y repetirlo por cada carpeta de nodo (sustituyendo "Node-1" por el nombre del resto de nodos), y ejecutar todos juntos, de forma que en menos de 1 segundo la tarea de eliminar esos archivos estará resuelta.

Capítulo 7 - Evaluación y comparativa

En este capítulo se describirá la evaluación de escalabilidad que se ha realizado, así como una comparativa con mecanismos de consenso ya existentes. Se ha evaluado el prototipo con un número limitado de nodos, pero se indican posibles problemas de escalabilidad en el caso de que el sistema estuviese formado por un número mucho mayor.

7.1 Escalabilidad

La prueba realizada consistió en la creación de una red formada por 10 nodos, con una ronda de votación cada 5 bloques, y un máximo de 5 validadores, funcionando correctamente la sincronización de estos y el sistema de votación.

A continuación, se muestra un ejemplo de proceso de votación. En un proceso de votación participan todos los nodos de la red, y cada uno emite un voto para seleccionar aquellos nodos que pasarán a formar parte de la lista de validadores, que son los nodos que pueden generar nuevos bloques en la cadena de bloques. En este ejemplo particular, los 10 nodos que forman la red pueden seleccionar hasta 5 validadores. En la Tabla 3 se muestran, para cada nodo, el número de nodo al que vota. Por ejemplo, en la primera columna, el nodo 1 vota por el nodo 4, y en la segunda, el nodo 2 no ha emitido ningún voto. La tercera fila contiene el valor de reputación de cada uno de los nodos de la red.

Tabla 3 – Esquema de votación de la prueba con 10 nodos

Nodos	1	2	3	4	5	6	7	8	9	10
Nodo votado	4	-	8	8	3	2	2	3	4	1
Reputación	56	100	12	22	22	22	22	42	22	32

Los resultados de la votación se muestran en la Tabla 4. Una vez emitidos los votos, el contrato de reputación realiza el recuento de votos, que aparece en la segunda fila de la tabla. Los votos se ponderan con el resultado que se muestra en la tercera fila. Finalmente, los nodos seleccionados son los que aparecen en la última fila, ordenados por el número indicado.

Tabla 4 – Resultados de la votación en la prueba con 10 nodos

Nodos	1	2	3	4	5	6	7	8	9	10
Recuento votos	1	2	2	2	0	0	0	2	0	0
Votos ponderados	22	44	64	77	0	0	0	34	0	0
Seleccionados (max. 5)	1		4	3				2		

Como se puede observar, el nodo 2 no fue elegido como validador, debido a que nunca había votado anteriormente (su *nonce* era cero). Además, como solo se votaron 5 nodos diferentes, y el nodo 2 fue descartado, solo 4 nodos resultaron seleccionados (de un máximo de 5).

El prototipo desarrollado en este trabajo nos permite estudiar los posibles problemas de escalabilidad que puede encontrar un mecanismo de consenso basado en reputación en un escenario real con un gran número de nodos.

El principal problema que se ha detectado que puede darse con un mayor número de nodos es con el algoritmo de ordenación del *Smart Contract*, cuando se utiliza para ordenar la lista de candidatos por sus votos recibidos. Se ha detectado que Hyperledger Besu revierte las transacciones que ejecutan métodos con alto consumo de gas. Esto ha ocasionado problemas durante el desarrollo, primero con cualquier

número de nodos, cuando se utilizaba el algoritmo de ordenación recursivo *QuickSort*, y después, ya con el algoritmo iterativo *BubbleSort*, cuando se hacían pruebas con un número de nodos mayor que 8. Finalmente, se encontró una solución provisional a este problema, que fue ordenar la lista de candidatos en el método *voteValidator* (de forma que se ordena automáticamente cada vez que un nodo envía su voto), en lugar de en el método *finishVoting*(), que tiene un consumo de gas mucho mayor por la ejecución de otros métodos, en comparación al primer método mencionado.

En el caso de ordenación de la lista de validadores por reputación, en la etapa de cierre de votación, el algoritmo de ordenación no sería tan problemático, porque la cantidad de validadores siempre es acotada, mientras que la cantidad de candidatos puede crecer de forma indefinida.

7.2 Seguridad

Con respecto a la seguridad, se han tenido en cuenta diversos aspectos que se detallan a continuación.

En el contrato se han incluido múltiples validaciones de transacciones (con el uso de modifiers de Solidity), que son las siguientes:

- *Modifier* ***isOwner****()*: comprueba que el *address* que ejecuta la transacción es el dueño del contrato (el nodo que lo desplegó). Se utiliza en el método *deleteValidator()* para solamente permitir que el *owner* pueda eliminar un validador de la lista, y en los métodos de actualización de los pesos y del número máximo de validadores: *setWeightBalance()*, *setWeightNonce()*, *setWeightBlocks()* y *setMaxValidators()*.
- *Modifier* ***isAllowed****()*: comprueba que el nodo es validador, o lo era en el momento que llamó al método (para el caso especial de que un nodo produzca el bloque de cierre de votación y en ese momento deje de ser validador). Se utiliza en el método *nextTurn()*, que lo ejecutan los validadores justo después de producir un bloque.
- *Modifier* ***hasCorrectProxyAddress****()*: comprueba que el contrato con el *address* indicado tiene el mismo *address* de proxy que el contrato actual.

Se utiliza en el método *updateContractAddress()* para comprobar que el contrato que hay en el *address* indicado no es malicioso.

- *Modifier* ***isContract****():* comprueba que el *address* indicado es un contrato. Se utiliza en el método *updateContractAddress()* para comprobar que el *address* es correcto y pertenece a un *Smart Contract*. Se ha usado la librería "Address" de *Openzeppelin* (21) para comprobar si un *address* es un contrato, usando el método *isContract(),* para evitar que se intente actualizar el address del contrato con uno que no sea válido.
- *Modifier* ***notVotedYet****()*: comprueba que el *address* del nodo votantes no está en la lista de votantes. Se utiliza en el método *voteValidator()* para asegurar que un nodo no pueda votar varias veces y evitar votos duplicados.
- *Modifier* ***notVoteHimself****()*: comprueba que el *address* indicado no coincide con el nodo votante. Se utiliza en el método *voteValidator()* para asegurar que un nodo no pueda votarse a sí mismo.
- *Modifier* ***isVotingRound****()*: comprueba que el bloque actual coincide con la ronda de votación. Se utiliza en el método *voteValidator()* para evitar que los nodos voten fuera de tiempo.
- *Modifier* ***notInBlackList****()*: comprueba que el *address* indicado no se encuentra en la *black list.* Se utiliza en el método *voteValidator()* para evitar que los nodos que están en la lista negra puedan votar.

En el sistema de votación se ha incluido una lista negra de validadores, a la cual se añaden nodos que hayan enviado un *nonce* erróneo (menor o igual que el anterior, o cero en el caso inicial), lo que significa que han modificado el código fuente de Hyperledger Besu para conectarse a la cadena de bloques. Estos nodos añadidos a la lista negra no podrán volver a votar ni ser elegidos validadores.

En el método *addValidators*(), que se encarga de añadir los nuevos validadores tras el periodo de votación, también se realizan validaciones por seguridad, como ya se mencionó anteriormente. Estas son las siguientes: no se podrá nunca añadir un nodo como validador si el balance de este es 0, ya que no podría no ejecutar transacciones;

tampoco si su *nonce* es 0, lo que significaría que nunca ha votado, y no se podría calcular su reputación correctamente (faltaría un factor); ni, por supuesto, si su *address* está en la lista negra).

Por otra parte, en Hyperledger Besu existen una serie de reglas de validación a las que se someten los nuevos bloques importados, con el fin de detectar si un bloque es malicioso o incorrecto, y no añadirlo a la *blockchain*.

En el código de Besu también se ha tenido en cuenta el posible uso excesivo de recursos del sistema, ya que en el proceso de producir un nuevo bloque todos los nodos se quedan en espera por un tiempo, y para evitar el consumo excesivo por espera activa, se duermen los hilos durante 0.1s con *Thread.sleep (100ms)*.

En cuanto al periodo de votación, en principio no es modificable una vez se ha creado la *blockchain*. Solo sería posible modificarlo desplegando un nuevo contrato y actualizando a una nueva versión de Besu que contenga el mismo periodo de votación, por seguridad. Si se hubiera permitido modificarlo con un método en el contrato, esto podría llevar a incompatibilidades con el cliente de Besu, en el caso de que la variable *votingRound* no coincidiese en ambos.

En cuanto a la actualización del *address* del contrato en el contrato Proxy, solo es posible mediante una llamada desde el propio contrato de consenso actual (el método *setConsensusAddress*() está restringido a ser llamado por la actual *consensusAddress*). A su vez, el método *updateContract*(), que llama al método de ProxyContract, solo puede ser ejecutado por el *owner* del contrato, por seguridad.

7.3 Comparativa

En esta sección se van a realizar dos comparaciones del mecanismo de reputación implementado, Repu, con el sistema de partida Clique, y con otro mecanismo basado en *Proof of Authority* contenido también en Hyperledger Besu, QBFT.

Comparando Repu con el mecanismo Clique, del que se realizó un clonado total y se modificó su funcionalidad, podríamos decir que se ha "desprendido" de la lógica en Java para seleccionar a los siguientes validadores.

Si bien la lógica de sincronización entre nodos (importación de bloques producidos) es común entre ambos, la funcionalidad relativa a la selección de siguientes validadores de bloques ha cambiado por completo. En Repu se ha creado un *Smart Contract* que se despliega en los primeros tres bloques y se utiliza para la gestión total del mecanismo de consenso. Es el contrato el que decide quien debe validar los siguientes bloques y en qué orden. Esta información es pública para cualquier que realice una consulta a este contrato. Esto aporta transparencia, fiabilidad y elegancia al mecanismo de consenso.

También se ha implementado en Repu un sistema de votación basado en reputación, que pondera los votos de cada nodo en función de su reputación en cada momento. Este mecanismo también está completamente implementado en el contrato, si bien desde Hyperledger Besu en Java se hacen las llamadas necesarias para que el sistema funcione.

Se podría decir que se ha sustituido la lógica que existía en Clique para seleccionar validadores a través de clases implementadas en *Web3j* que se conectan con un *Smart Contract*, y es dicho contrato quien contiene toda esa lógica (con las novedades del sistema basado en reputación).

Si comparamos Repu con el mecanismo QBFT, se aprecian mayores similitudes. QBFT es un mecanismo basado en *Proof of Authority* que tiene un sistema de votación para validar los bloques previamente a ser añadidos a la *blockchain*. Cada bloque debe ser aprobado por 2/3 de los validadores existentes.

Además, tiene dos sistemas para seleccionar validadores, uno programado en Java (al estilo de Clique) y otro implementado en un *Smart Contract* pre-desplegado en la *blockchain* en el bloque génesis, que también funciona mediante un sistema de votación.

Aunque Repu no utiliza su sistema de votación para aprobar cada bloque (ya que el sistema de partida, Clique, carecía de esa funcionalidad), sí que se utiliza también un *Smart Contract* con un sistema de votación para seleccionar a los validadores, al igual que en QBFT.

Sin embargo, en Repu no se logró pre-desplegar los contratos, y se utilizan tres bloques iniciales "génesis" para ello. Por otra parte, en Repu se ha implementado la posibilidad de actualizar el contrato de reputación, gracias a contrato Proxy. Esto último es una ventaja frente a QBFT, cuyo contrato no es modificable una vez creada la *blockchain*.

Si comparamos el *Smart Contract* de Repu con el contrato utilizado por QBFT (22), encontramos variables y métodos similares.

También tienen un número máximo de validadores (en su caso no es modificable, a diferencia de Repu), utilizan *mappings* y *arrays* para gestionar los validadores y los votos, y tienen métodos para obtener los validadores y gestionar el sistema de votación.

Lo más destacable de Repu respecto a QBFT es que incorpora la novedad de un mecanismo de consenso basado en reputación utilizando un *Smart Contract*, algo que hasta el momento no se ha desarrollado en ningún proyecto conocido.

Capítulo 8 - Conclusiones y trabajo futuro

En esta sección se va a hacer un balance entre los objetivos planteados al inicio y los logros conseguidos, a modo de conclusión, y también se mencionarán posibles mejoras/nuevas funcionalidades que se podrían desarrollar como trabajo futuro en el proyecto.

El primer objetivo, "estudiar mecanismos de consenso alternativos", se ha cumplido estudiando diferentes artículos académicos sobre *Proof of Reputation*, como se describe en la sección "Estado del Arte".

El segundo objetivo, "estudiar la posibilidad de implementar un mecanismo de consenso programado en un *Smart Contract*", también se ha logrado con éxito, si bien han surgido limitaciones relacionadas con el algoritmo de ordenación.

El tercer objetivo, "Implementar un prototipo", se ha conseguido alcanzar y también se ha estudiado su escalabilidad.

El cuarto objetivo, "estudiar qué nodo Ethereum utilizar", también ha sido un éxito, pues se ha utilizado el nodo Hyperledger Besu y se ha conseguido implementar el prototipo partiendo de él.

El quinto y último objetivo marcado, "Hacer pruebas con una red de nodos local, y estudiar escalabilidad", también se ha cumplido. Como se ha descrito en la sección de escalabilidad anteriormente, se han realizado con éxito pruebas con 10 nodos.

A continuación, se describirá el posible trabajo futuro.

Queda pendiente la implementación de la funcionalidad de un *time-out* para que, en el caso en el que un nodo no responda, permitir que el siguiente nodo sea quien valide. Sin embargo, esto no es fácil de implementar, debido a que el mecanismo de sincronización de nodos está integrado en el *core* de Hyperledger Besu, y cambiar esto conllevaría modificar el funcionamiento de otros mecanismos de consenso, lo cual es problemático. La solución ideal sería crear una clase específica para esta funcionalidad dentro de Repu.

Se intentó implementar el *time-out*, llegando a una versión primitiva, pero finalmente se descartó por falta de tiempo. Esta implementación funcionaba en el caso de que el validador que no responde no vuelve a conectarse, y el resto de validadores esperan 30 segundos desde el tiempo de producción del último bloque para pasar al siguiente validador de la lista si no se ha producido el siguiente bloque. Este código está comentado en la clase RepuHelpers, como se muestra en la figura 8-1

```
if (Objects.equals(candidate.toString(), validations.get(String.valueOf( i: parent.getNumber() + 1)))) {
    return true;
} else {
    /*List<String> nextValidators = repuContract.nextValidators();
    long lastTimestamp = parent.getTimestamp() * 1000;
    int i = 0;
    while (true) {
        if (System.currentTimeMillis() - lastTimestamp >= 30000) {
            i++;
            if (candidate.toString().equals(nextValidators.get(i % nextValidators.size()))) {
                validations.put(String.valueOf(parent.getNumber() + 1), candidate.toString());
                return true;
            }
        } else if (repuContract.getBlock() > parent.getNumber()) {
            break;
        } else {
            Thread.sleep(1000);
        }
    }*/
    return false;
}
```

Figura 8-1 Código comentado del prototipo de time-out implementado en Besu

Como posible mejora de seguridad se podría liberar a los nodos de la llamada al método "*nextTurn()*" del contrato, que en este momento es requerida para que la *blockchain* progrese, y podría ser un agujero de seguridad si alguien modificase el código fuente. Para mejorar este aspecto, se podría intentar ejecutar automáticamente en el contrato.

También queda pendiente como trabajo futuro implementar los *tests* del mecanismo Repu correctamente, que por el momento son un clonado de los de Clique.

Como se explicó en apartados anteriores, en teoría es posible pre-desplegar *Smart Contracts* en el bloque génesis (primer bloque), ya que así está descrito en la documentación de Hyperledger Besu (15). Sin embargo, por la razón de que dicha

documentación es muy pobre, y por falta de tiempo, no se logró desplegar correctamente ni siquiera un contrato simple. Con las pruebas realizadas se consiguió desplegar un contrato, pero este no respondía correctamente, sino que devolvía datos en hexadecimal desconocidos.

Una posible mejora futura sería investigar más ese aspecto, para conseguir pre-desplegar al menos el contrato Proxy, y si es posible también el contrato de consenso RepuContract. De esa forma, se conseguiría reducir los actuales tres "bloques génesis" a dos, o, en el mejor de los casos, a uno.

Otra posible mejora sería crear un contrato de *stake*, al estilo del de la red Ethereum, con el fin de sustituir el actual factor de reputación "balance" por uno más justo y fiable, que sería la cantidad de dinero en *stake*, es decir, la cantidad de dinero apostada por un nodo, al igual que en las redes que utilizan *Proof of Stake*.

Otra posible mejora sería implementar un factor de aleatoriedad en la reputación, con el fin de evitar que una parte de la red tenga siempre el control. Sin embargo, esta opción se descartó en principio por los problemas de seguridad que ocasionan los números aleatorios en Solidity (las operaciones dentro de la cadena de bloques deben ser deterministas, es decir, producir el mismo resultado independientemente de quien haga la llamada). Se puede utilizar un mecanismo similar al de Ethereum, que utiliza un factor de aleatoriedad en su mecanismo de consenso *Proof of Stake*. Como trabajo futuro, se podría investigar cómo gestionan este asunto en Ethereum y tratar de implementar su solución en Repu.

Otro aspecto que se podría mejorar en el futuro es la descentralización del poder de control sobre el *Smart Contract*. En este momento, ciertas acciones sobre el contrato solo pueden ser llevadas a cabo por el *owner*, es decir, el nodo que lo desplegó inicialmente, y esto es, en cierto modo, un problema de seguridad, teniendo en cuenta que el *owner* puede dejar de ser validador como un nodo más y podría tomar represalias mediante la modificación de valores del contrato. Por ejemplo, actualizar el *address* del propio contrato, eliminar validadores, actualizar el número máximo de validadores o actualizar los *weights* de los factores de reputación. Una posibilidad para descentralizar totalmente la gestión del contrato sería utilizar una DAO (*Decentralized*

Automous Organization), de formar que haya varios nodos *owner* y se realicen votaciones para las tomas de decisiones sobre el contrato.

Otra posible mejora sería enviar más *feedback* a los nodos desde el contrato sobre el resultado de ejecución de los diferentes métodos. Por ejemplo, si un candidato a validador ha sido rechazado en el proceso de validación, especificar la razón de ello.

Como se comentó anteriormente, como trabajo futuro y necesario para la escalabilidad del prototipo, habría que investigar una solución eficiente y segura para ordenar los *arrays* del *Smart Contract*, para evitar un excesivo consumo de gas por la ejecución de métodos en este, ni tampoco comprometer la seguridad de la red realizando la ordenación en el código fuente de los nodos, que es manipulable.

Introduction

Motivation

Blockchain is a technology that creates a public and secure record of transactions and data, in a decentralized and distributed manner, by storing them in "blocks" that contain a hash reference to the previous block, thus forming an immutable chain. (This concept will be explained in more detail in section 2). (1) (2) (3)

The immutability of the data stored in the blocks of the chain is guaranteed by the use of cryptographic hash functions, since a hash function is an algorithm which produces one set of characters for an input, and which, if the input is modified, produces a completely different set. Thanks to this feature, if the information in the blocks were to be modified, it would be easily detected.

In addition to this feature of immutabilities, blockchain also guarantees the authenticity of the stored data, since each of the transactions carried out is digitally signed by the issuers using asymmetric cryptographic keys. This ensures that it is not possible to impersonate the issuer or alter transactions once they have been carried out.

However, the blockchain system, being a distributed and decentralized network, requires a mechanism that allows the different nodes of the network to collaborate and remain secure. To do this, all nodes in the network must agree on who should produce the next block and the validity of its content. This is known as a consensus mechanism, and is used to produce new blocks on the blockchain, i.e. all nodes in the network must agree on who should produce the next block.

There are currently multiple consensus mechanisms, the best known being Proof of Work (from the Bitcoin network), Proof of Stake (from the Ethereum network) and Proof of Authority (from private networks). Each of them uses different concepts and mechanisms to reach consensus between nodes, but the aim is the same: to select by consensus which node will produce the next block.

This work is based on the idea of creating a prototype of an innovative Blockchain network consensus mechanism, taking as a central idea the reputation-based systems proposed in several academic articles (4) (5) (6). A consensus mechanism based on reputation systems would allow the criteria used to choose the next validator (node that will produce the next block) to be adjusted more dynamically than other mechanisms. It could be used in private or permissioned networks, or even in majority networks such as Bitcoin or Ethereum.

As a starting point, one can take an existing Ethereum node, such as Geth (7), programmed in Go, or Hyperledger Besu (8), programmed in Java, which allow the creation of private Blockchain networks, ideal for testing this type of consensus mechanism.

Goals

The main objective of this work is to study the characteristics of a blockchain system that uses a consensus mechanism based on Proof of Reputation (PoR). To this end, we will implement a prototype that will allow us to evaluate the most relevant aspects of the system, using an existing open-source implementation of Ethereum as a starting point. Specifically, the specific objectives of this work are the following:

- To study alternative consensus mechanisms to the most popular ones, focusing especially on Proof of Reputation.
- To study the possibility of implementing a consensus mechanism programmed in a Smart Contract, delegating all the management of the mechanism to it and freeing the nodes from this functionality. This would make it possible to have a secure, transparent, traceable and updatable consensus mechanism.
- Study the applicability of reputation systems to the consensus mechanisms of blockchain systems. Design a PoR (Proof of Reputation) consensus mechanism.
- Study the current implementations of Ethereum blockchain systems to choose the most suitable one for the implementation of a PoR consensus prototype.
- Once the prototype is implemented, test it by setting up a local network of nodes, and see to what extent it would be scalable in a real network.

Work plan

During the development of this work, weekly meetings have been held between tutor and student for the monitoring and planning of the following tasks:

1.Study the existing Ethereum implementations and decide which one is the most suitable to use as the basis for this work.

2.Study the existing consensus mechanisms in the node and see which one would be most similar to Proof of Reputation.

3.Study in detail the consensus proposals based on Proof of Reputation (PoR) published in academic articles.

4.Set up a test blockchain network with one or two nodes to learn how the chosen Ethereum implementation works and compile and run the source code of that implementation.

5.Design a basic PoR scheme based on the academic proposals studied in point 3. Evaluate the feasibility of using smart contracts to implement the PoR consensus mechanism.

6.Implement the design, either with a Smart Contract or in the node's own management program, as designed in the previous step.

7.Test the implemented prototype in order to compare it with existing mechanisms and test scalability.

8.Document the project by framing the alternative mechanism chosen in the existing algorithms.

Conclusions and future work

This section will take stock of the objectives set at the beginning and what has been achieved, by way of conclusion, and will also mention possible improvements/new functionalities that could be developed as future work on the project.

The first objective, "to study alternative consensus mechanisms", has been fulfilled by studying different academic articles on Proof of Reputation, as described in the "State of the Art" section.

The second objective, "to study the possibility of implementing a programmed consensus mechanism in a Smart Contract", has also been successfully achieved, although limitations related to the sorting algorithm have arisen.

The third objective, "To implement a prototype", has been achieved and its scalability has also been studied.

The fourth objective, "to study which Ethereum node to use", has also been a success, as the Hyperledger Besu node has been used and the prototype has been implemented based on it.

The fifth and last objective, "To test with a local node network and study scalability", has also been achieved. As described in the scalability section above, tests have been successfully carried out with 10 nodes.

Possible future work will be described below.

It remains to implement a time-out functionality so that, in the case where a node does not respond, the next node is allowed to validate. However, this is not easy to implement, because the node synchronisation mechanism is integrated in the core of Hyperledger Besu, and changing this would imply modifying the functioning of other consensus mechanisms, which is problematic. The ideal solution would be to create a specific class for this functionality within Repu.

An attempt was made to implement the time-out, reaching a primitive version, but it was finally discarded due to lack of time. This implementation worked in the case that the validator that does not respond does not reconnect, and the rest of the

validators wait 30 seconds from the production time of the last block to pass to the next validator in the list if the next block has not been produced. This code is commented out in the RepuHelpers class, as shown in Figure 8-1.

```
if (Objects.equals(candidate.toString(), validations.get(String.valueOf( parent.getNumber() + 1)))) {
    return true;
} else {
    /*List<String> nextValidators = repuContract.nextValidators();
    long lastTimestamp = parent.getTimestamp() * 1000;
    int i = 0;
    while (true) {
        if (System.currentTimeMillis() - lastTimestamp >= 30000) {
            i++;
            if (candidate.toString().equals(nextValidators.get(i % nextValidators.size()))) {
                validations.put(String.valueOf(parent.getNumber() + 1), candidate.toString());
                return true;
            }
        } else if (repuContract.getBlock() > parent.getNumber()) {
            break;
        } else {
            Thread.sleep(1000);
        }
    }*/
    return false;
}
```

Figura 8-2 Annotated code of the time-out prototype implemented in Besu

As a possible security improvement, nodes could be freed from calling the contract's "*nextTurn*()" method, which is currently required for the blockchain to progress, and could be a security hole if someone were to modify the source code. To improve this aspect, one could try to execute automatically in the contract.

Future work also remains to be done to implement the tests of the Repu mechanism correctly, which for the moment are a clone of those of Clique.

As explained in previous sections, in theory it is possible to pre-deploy Smart Contracts in the genesis block (first block), as this is described in the Hyperledger Besu documentation (15). However, due to the poor documentation and lack of time, it was not possible to correctly deploy even a simple contract. With the tests performed, it was possible to deploy a contract, but it did not respond correctly, but returned data in unknown hexadecimal.

A possible future improvement would be to investigate this aspect further, in order to pre-deploy at least the proxy contract, and if possible, also the consensus contract RepuContract. This would reduce the current three "genesis blocks" to two, or at best one.

Another possible improvement would be to create a stake contract, in the style of the Ethereum network, in order to replace the current reputation factor "balance" with a fairer and more reliable one, which would be the amount of money in stake, i.e. the amount of money bet by a node, as in networks that use Proof of Stake.

Another possible improvement would be to implement a randomization factor in reputation, in order to avoid one part of the network always being in control. However, this option was initially ruled out because of the security problems caused by random numbers in Solidity (operations within the blockchain must be deterministic, i.e. produce the same result regardless of who makes the call). A mechanism similar to Ethereum, which uses a randomness factor in its Proof of Stake consensus mechanism, can be used. As future work, one could investigate how they handle this issue in Ethereum and try to implement their solution in Repu.

Another aspect that could be improved in the future is the decentralization of control power over the Smart Contract. At the moment, certain actions on the contract can only be carried out by the owner, i.e. the node that initially deployed it, and this is, in a way, a security problem, taking into account that the owner can stop being a validator as just another node and could retaliate by modifying contract values. For example, updating the address of the contract itself, removing validators, updating the maximum number of validators or updating the weights of the reputation factors. One possibility to fully decentralized contract management would be to use a DAO (Decentralized Autonomous Organization), so that there are several owner nodes and voting takes place for contract decisions.

Another possible improvement would be to send more feedback to the nodes from the contract on the result of the execution of the different methods. For example, if a candidate validator has been rejected in the validation process, specify the reason for this.

As mentioned above, as future work necessary for the scalability of the prototype, it would be necessary to investigate an efficient and secure solution to sort the Smart Contract arrays, to avoid excessive gas consumption by the execution of methods in it, nor to compromise the security of the network by performing the sorting in the source code of the nodes, which is manipulable.

BIBLIOGRAFÍA

1. **Bit2me Academy.** Bit2me Academy. *¿Qué es Blockchain, la tecnología de la Cadena de Bloques?* [En línea] https://academy.bit2me.com/que-es-cadena-de-bloques-blockchain/.

2. **Buterin, Vitalik.** Ethereum White Paper: A Next Generation Smart Contract \& Decentralized Application Platform. [En línea] 2014. https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum_Whitepaper_-_Buterin_2014.pdf.

3. **Nakamoto, Satoshi.** Bitcoin: A Peer-to-Peer Electronic Cash System. [En línea] 2009. https://bitcoin.org/bitcoin.pdf.

4. *Proof-of-Reputation: An Alternative Consensus Mechanism for Blockchain Systems.* **Aluko, Oladotun and Kolonin, Anton.** 2021.

5. *Delegated Proof of Reputation: A Novel Blockchain Consensus.* **Thuat Do et al.** 10.1145/3343147.3343160 : Association for Computing Machinery, 2019. IECC '19: Proceedings of the 1st International Electronics Communication Conference. pág. 9.

6. *Proof of Reputation: A Reputation-based Consensus Protocol for Blockchain Based Systems.* **Zhuang, Qianwei, et al.** New York, NY, USA : Association for Computing Machinery, 2019. IECC '19: Proceedings of the 1st International Electronics Communication Conference. pág. 8.

7. **Go-Ethereum.** Getting started with Geth. [En línea] https://geth.ethereum.org/docs/getting-started.

8. **Hyperledger Foundation.** Hyperledger Besu. [En línea] https://www.hyperledger.org/use/besu.

9. **ethereum.org.** Consensus. [En línea] https://ethereum.org/en/developers/docs/consensus-mechanisms/.

10. *The Bizantine Generals Problem.* **Lamport, Leslie et al.** 1982. ACM TOPLAS.

11. Bitcoin Gold Blockchain Hit by 51% Attack Leading to $70K Double Spend. [En línea] 2018. **Martin, Jack.** https://cointelegraph.com/news/bitcoin-gold-blockchain-hit-by-51-attack-leading-to-70k-double-spend.

12. **ethereum.org.** Proof of Work (PoW). [En línea] https://ethereum.org/es/developers/docs/consensus-mechanisms/pow/.

13. **Mena Roa, Mónica**. Bitcoin consume más electricidad que países enteros. [En línea] 2021. https://es.statista.com/grafico/18630/consumo-de-electricidad-anual-de-bitcoin/.

14. Linux Foundation. [En línea] https://www.linuxfoundation.org/.

15. **Hyperledger Besu**. Pre-deploy contracts in the genesis file. [En línea] https://besu.hyperledger.org/en/stable/private-networks/how-to/configure/contracts/?h=genesis+storage.

16. **Web3j**. [En línea] https://docs.web3j.io/4.10.0/.

17. **Solidity**. Installing the Solidity Compiler. [En línea] https://docs.soliditylang.org/en/v0.8.17/installing-solidity.html#installing-the-solidity-compiler.

18. **Wikipedia**. Ordenamiento de burbuja. [En línea] https://es.wikipedia.org/wiki/Ordenamiento_de_burbuja.

19. **Hyperledger Besu**. Create a private network using Clique. [En línea] https://besu.hyperledger.org/en/stable/private-networks/tutorials/clique/.

20. **Hyperledger Besu**. Configure Clique consensus. [En línea] https://besu.hyperledger.org/en/stable/private-networks/how-to/configure/consensus/clique/.

21. **Openzeppelin**. [En línea] https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/Address.sol.

22. **ConsenSys**. **Github**. *ValidatorSmartContractAllowList.sol.* [En línea] https://github.com/ConsenSys/validator-smart-contracts/blob/main/contracts/allowlist/ValidatorSmartContractAllowList.sol.

23. **Razorblissrotation**. Theymos from Bitcoin wikivectorization. [En línea]

24. **Hyperledger Besu**. Hyperledger Besu architecture. [En línea] https://besu.hyperledger.org/en/22.1.3/Concepts/ArchitectureOverview/.