Proyecto – Gobernanza y sistemas de votación

1. *DAOs* y sistemas de votación *on chain*

Las organizaciones autónomas descentralizadas (*Decentralized Autonomous Organizations*, DAO)[1] se definen por unas reglas públicas de gobernanza. La necesidad de que las reglas sean públicas e inmodificables y que cualquiera pueda comprobar que todas las acciones realizadas han seguido las reglas, requiere un nivel de transparencia que, quizá, solo la tecnología blockchain nos pueda proporcionar. Las reglas de gobernanza se describen en forma de código en un smart contract que es público e inmodificable una vez cargado en el blockchain y todas las acciones realizadas (transacciones) son registradas en el blockchain y están disponibles para cualquier usuario.

Este modelo evita muchos de los problemas de la gobernanza fuera de un blockchain, donde las reglas no son claras y pueden cambiarse (o aplicarse mal) según decisiones de unos pocos y donde algunas de estas acciones pueden ocultarse al público.

Una de las formas más comunes de gobernanza en una DAO es por votación. Es decir, se establece algún mecanismo para que los miembros de la DAO indiquen sus preferencias sobre las acciones a realizar y finalmente se ejecutan aquellas acciones que, según las reglas de gobernanza, han recibido el apoyo suficiente.

Los mecanismos de votación en el blockchain pueden requerir una participación mínima de los miembros o un mínimo apoyo por parte de los miembros del DAO a una acción para permitir que esta se realice. Este mínimo apoyo se define como un umbral que puede depender del estado actual del DAO. Por otro lado, se debe definir cuál es valor de los votos emitidos. Así, este valor puede estar relacionado directamente con la cantidad de dinero o peso que tiene el miembro dentro del DAO o, por el contrario, considerar que el voto de todos los miembros tiene el mismo peso. Una opción sencilla que está en medio de las dos utiliza la votación cuadrática, en la que el coste para un miembro de realizar un voto adicional crece cuadráticamente. De esta forma los miembros con más dinero tienen más peso, ya que tiene capacidad para emitir más votos, pero el crecimiento cuadrático del coste de cada voto adicional desincentiva la emisión de demasiados votos. Existen mecanismos mucho más complejos para definir el valor o el coste de un voto, pero en este proyecto nos quedaremos con la votación cuadrática.

[1] `https://ethereum.org/en/dao/`
`https://coinmarketcap.com/alexandria/article/what-is-a-dao`

2. Sistema de votación cuadrática de propuestas en DAOs

El objetivo de este proyecto es la implementación de un sistema de votación cuadrático de propuestas (`https://en.wikipedia.org/wiki/Quadratic_voting`) *on chain* para DAOs.

El proceso de votación se realiza durante un periodo de tiempo que suele durar varios días o semanas. Un usuario autorizado es el encargado de abrir y cerrar el periodo de votación (utilizaremos la dirección del creador del contrato de la votación para identificarle); ningún otro participante puede realizar esta tarea. Este usuario es el que recibirá la parte no gastada del presupuesto de la votación cuando termine el periodo de votación. Al abrir un periodo de votación se debe proporcionar un importe inicial en Ether para financiar las propuestas aprobadas. Este presupuesto se podrá incrementar con las aportaciones en *tokens* de los votos de las propuestas que se vayan aprobando durante el periodo de votación.

Los participantes pueden inscribirse en cualquier momento, incluso si ya se ha abierto el proceso de votación. Los participantes pueden añadir propuestas durante el periodo de votación. Tanto al inscribirse como posteriormente un votante puede transferir Ether para comprar *tokens* con los que depositar votos. El Ether invertido en *tokens* para votar una propuesta aprobada se utilizará para incrementar el presupuesto asignado, o bien se puede reintegrar este Ether a los participantes al final del periodo de votación si su propuesta no ha sido aprobada.

Durante el periodo de votación los participantes pueden depositar votos sobre las propuestas que deseen. Cada voto tiene un precio que se mide en *tokens*. Un participante puede votar a varias propuestas e incluso depositar varios votos sobre una misma propuesta, pero en este caso el precio de los votos crece cuadráticamente respecto al número de votos ya emitidos por el votante para esa propuesta. De esta forma, el primer voto a esa propuesta cuesta 1 *token*, pero si el mismo votante deposita un segundo voto a esa propuesta, el coste de los dos votos pasa a ser de 4 *tokens*, tres votos cuestan 9 *tokens*, y así sucesivamente.

Por ejemplo, si en un proceso de votación hay 16 propuestas y un votante dispone de un crédito de 16 *tokens*, puede depositar un voto a cada una de las 16 propuestas (cada voto vale 1 *token*), o bien puede depositar 4 votos a una sola propuesta por 16 *tokens*, o cualquier combinación intermedia (por ejemplo, dos votos a cuatro propuestas).

Hay dos tipos de propuestas que se pueden votar:

- **Propuestas de "*signaling*":** estas propuestas se pueden votar pero no tienen presupuesto asociado. Sirven para mostrar preferencias de enfoque del DAO. Cuando se votan estas propuestas, los *tokens* utilizados se retienen mientras dura el periodo de votación, y se devuelven a sus propietarios una vez que el periodo de votación termine.

- **Propuestas de financiación:** En este caso sí tienen presupuesto, de forma que, si el número de votos recibidos llega a un umbral determinado, se aprueba la propuesta. No es necesario que termine el periodo de votación para aprobar una propuesta: en cuanto se supere el umbral se debe aprobar la propuesta de financiación.

Una propuesta i es aprobada si se cumplen dos condiciones: (1) el presupuesto del contrato de votación más el importe recaudado por los votos recibidos es **suficiente para financiar la**

propuesta; y (2) el número de votos recibidos **supera un umbral** definido por la siguiente fórmula:

$$threshold_i = (0{,}2 + \frac{budget_i}{totalbudget}) \cdot numParticipants + numPendingProposals$$

donde:

- $budget_i$ es el presupuesto de la propuesta i,
- $totalbudget$ es el presupuesto total disponible para este proceso de votación en el momento de evaluar el umbral,
- $numParticipants$ es el número de participantes en el momento de evaluar el umbral, y
- $numPendingProposals$ es el número de propuestas de financiación no aprobadas ni canceladas en el momento de evaluar el umbral (las propuestas de *signaling* no se incluyen aquí).

Cada propuesta tendrá un valor de umbral diferente, que además puede cambiar a lo largo del periodo de votación. Observa que el cálculo del umbral puede variar cuando se producen diversas circunstancias. **Sin embargo, solo se debe recalcular el umbral de una propuesta cada vez que reciba votos.**

En cuanto se detecta que una propuesta cumple las condiciones para ser aprobada, el contrato del sistema de votación **ejecuta la propuesta**, que implica transferir al contrato de la propuesta el importe presupuestado mediante una llamada a un contrato externo que implementa el interfaz `IExecutableProposal`. Además debe actualizar la propuesta como "aprobada" y modificar el presupuesto total disponible, entre otras cosas. Consulta la descripción de `openVoting` en la siguiente sección para tener información más detallada.

Debes decidir los contratos que vas a implementar, que pueden ser varios, pero al menos debes implementar los que se indican a continuación:

2.1. Contratos de propuestas e interfaz `IExecutableProposal`

Las propuestas se representan mediante contratos que deben implementar el interfaz `ExecutableProposal`:

```
interface IExecutableProposal {
  function executeProposal(uint proposalId, uint numVotes,
    uint numTokens) external payable;
}
```

Cuando se añade una propuesta para ser votada, se debe proporcionar la dirección de un contrato que implemente este interfaz. Cuando se aprueba la propuesta, se debe ejecutar la función `executeProposal` sobre esa dirección, enviando en la transacción que la ejecute la cantidad de Ether presupuestada para esa propuesta.

Es importante tener en cuenta que estos contratos de propuesta son externos al sistema de votación. El sistema de votación solo recibe un address de un contrato que debe implementar este interfaz. Para probar el sistema, puedes hacer un pequeño contrato de pruebas que emita un evento cada vez que se llame a esta función que incluya el saldo de ese contrato para verificar que efectivamente ha recibido el dinero en la transacción.

2.2. Contrato `QuadraticVoting`

El contrato `QuadraticVoting` es el contrato que gestiona la votación. Debe incluir como mínimo las siguientes funciones:

- En la creación de este contrato se debe proporcionar el precio en Wei de cada *token* y el número máximo de *tokens* que se van a poner a la venta para votaciones. Entre otras cosas, el constructor debe crear el contrato de tipo ERC20 que gestiona los *tokens*. En la sección 3 se proporcionan más detalles sobre el código del estándar ERC20.

- **`openVoting`:** Apertura del periodo de votación. Solo lo puede ejecutar el usuario que ha creado el contrato. En la transacción que ejecuta esta función se debe transferir el presupuesto inicial del que se va a disponer para financiar propuestas. Recuerda que este presupuesto total se modificará cuando se aprueben propuestas: se incrementará con las aportaciones en *tokens* de los votos de las propuestas que se vayan aprobando y se decrementará por el importe que se transfiere a las propuestas que se aprueben.

- **`addParticipant`:** Función que utilizan los participantes para inscribirse en la votación. Los participantes se pueden inscribir en cualquier momento, incluso antes de que se abra el periodo de votación. Cuando se inscriben, los participantes deben transferir Ether para comprar *tokens* (al menos un *token*) que utilizarán para realizar sus votaciones. Esta función debe crear y asignar los *tokens* que se pueden comprar con ese importe.

- **`removeParticipant`:** Función para que un participante pueda eliminarse del sistema. Un participante que invoca esta función no podrá depositar votos, crear propuestas ni comprar o vender *tokens*, a no ser que se vuelva a añadir como participante.

- **`addProposal`:** Función que crea una propuesta. Cualquier participante puede crear propuestas, pero solo cuando la votación está abierta. Recibe todos los atributos de la propuesta: título, descripción, presupuesto necesario para llevar a cabo la propuesta (puede ser cero si es una propuesta de *signaling*) y la dirección de un contrato que implemente el interfaz `ExecutableProposal`, que será el receptor del dinero presupuestado en caso de ser aprobada la propuesta. Debe devolver un identificador de la propuesta creada.

- **`cancelProposal`:** Cancela una propuesta dado su identificador. Solo se puede ejecutar si la votación está abierta. El único que puede realizar esta acción es el creador de la propuesta. No se pueden cancelar propuestas ya aprobadas. Los *tokens* recibidos hasta el momento para votar la propuesta deben ser devueltos a sus propietarios.

- **`buyTokens`:** Esta función permite a un participante ya inscrito comprar más *tokens* para depositar votos.

- **`sellTokens`:** Operación complementaria a la anterior: permite a un participante devolver *tokens* no gastados en votaciones y recuperar el dinero invertido en ellos.

- **`getERC20`:** Devuelve la dirección del contrato ERC20 que utiliza el sistema de votación para gestionar *tokens*. De esta forma, los participantes pueden utilizarlo para operar con los *tokens* comprados (transferirlos, cederlos, etc.).

- **`getPendingProposals`:** Devuelve un array con los identificadores de todas las propuestas de financiación pendientes de aprobar. Solo se puede ejecutar si la votación está abierta.

- **`getApprovedProposals`:** Devuelve un array con los identificadores de todas las propuestas de financiación aprobadas. Solo se puede ejecutar si la votación está abierta.

- **`getSignalingProposals`:** Devuelve un array con los identificadores de todas las propuestas de *signaling* (las que se han creado con presupuesto cero). Solo se puede ejecutar si la votación está abierta.

- **`getProposalInfo`:** Devuelve los datos asociados a una propuesta dado su identificador. Solo se puede ejecutar si la votación está abierta.

- **`stake`:** recibe un identificador de propuesta y la cantidad de votos que se quieren depositar y realiza el voto del participante que invoca esta función. Calcula los *tokens* necesarios para depositar los votos que se van a depositar y comprueba que el participante ha cedido (con `approve`) el uso de esos *tokens* a la cuenta del contrato de la votación. Recuerda que un participante puede votar varias veces (y en distintas llamadas a `stake`) una misma propuesta con coste total cuadrático.

  El código de esta función debe transferir la cantidad de *tokens* correspondiente desde la cuenta del participante a la cuenta de este contrato `QuadraticVoting` para poder operar con ellos. Como esta transferencia la realiza este contrato, el votante debe haber cedido previamente con `approve` los *tokens* correspondientes a este contrato (esa cesión de *tokens* no se debe programar en `QuadraticVoting`: la debe realizar el participante con el contrato ERC20 antes de ejecutar esta función; el contrato ERC20 se puede obtener con `getERC20`).

- **`withdrawFromProposal`:** Dada una cantidad de votos y el identificador de la propuesta, retira (si es posble) esa cantidad de votos depositados por el participante que invoca esta función de la propuesta recibida. Un participante solo puede retirar de una propuesta votos que él haya depositado anteriormente y la propuesta no ha sido aprobada o cancelada. Recuerda que debes devolver al participante los *tokens* que utilizó para depositar los votos que ahora retira (por ejemplo, si había depositado 4 votos a una propuesta y retira 2, se le deben devolver 12 *tokens*).

- **`_checkAndExecuteProposal`:** Función **interna** que comprueba si se cumplen las condiciones para ejecutar una propuesta de financiación y si es así la ejecuta utilizando la función `executeProposal` del contrato externo proporcionado al crear la propuesta. En esta llamada debe transferirse a dicho contrato el dinero presupuestado para su ejecución. Recuerda que debe actualizarse el presupuesto disponible para propuestas (y no olvides añadir al presupuesto el importe recibido de los *tokens* de votos de la propuesta que se acaba de aprobar). Además deben eliminarse los *tokens* asociados a los votos recibidos por la propuesta, pues la ejecución de la propuesta los consume.

  Las propuestas de *signaling* no se aprueban durante el proceso de votación: se ejecutan todas cuando se cierra el proceso con `closeVoting`.

Cuando se realice la llamada a `executeProposal` del contrato externo, se debe limitar la cantidad máxima de gas que puede utilizar para evitar que la propuesta pueda consumir todo el gas de la transacción. Esta llamada debe consumir como máximo 100000 gas.

- **`closeVoting`:** Cierre del periodo de votación. Solo puede ejecutar esta función el usuario que ha creado el contrato de votación. Cuando termina el periodo de votación se deben realizar entre otras las siguientes tareas:
  - Las propuestas de financiación que no han podido ser aprobadas son descartadas y los *tokens* recibidos por esas propuestas es devuelto a sus propietarios.
  - Todas las propuestas de *signaling* son ejecutadas y los *tokens* recibidos mediante votos es devuelto a sus propietarios.
  - El presupuesto de la votación no gastado en las propuestas se transfiere al propietario del contrato de votación.

  Cuando se cierra el proceso de votación no se deben aceptar nuevas propuestas ni votos y el contrato `QuadraticVoting` debe quedarse **en un estado que permita abrir un nuevo proceso de votación.**

  Esta función puede consumir una gran cantidad de gas, tenlo en cuenta al programarla y durante las pruebas.

3. *Tokens* ERC20

Se deben utilizar *tokens* ERC20 para gestionar las votaciones. Además, la implementación de este estándar debe permitir crear nuevos *tokens* para los participantes. Para ello, debes implementar tu *token* ERC20 heredando del código disponible de este estándar en el repositorio de la empresa OpenZeppelin:

- La versión para Solidity 0.8.x está en:
  `https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/`
- La versión para Solidity 0.7.x está en:
  `https://github.com/OpenZeppelin/openzeppelin-contracts/tree/solc-0.7/contracts/token/ERC20`

Puedes encontrar más información en:

`https://docs.openzeppelin.com/contracts/4.x/api/token/erc20`

Algunas observaciones sobre esta implementación de ERC20:

- La documentación de OpenZeppelin indica que se importen los ficheros fuente directamente de internet (con sentencias del tipo `import "@openzeppelin/...";` donde el símbolo `@` indica al compilador que descargue el código del repositorio de `https://github.com/`). Tu contrato ERC20 debe heredar de la implementación de OpenZeppelin e implementar **lo mínimo imprescindible** para que funcione en tu sistema.

- Para crear nuevos *tokens* debes implementar una función externa que llame a la función interna `_mint` heredada de la implementación de OpenZeppelin. Puedes mirar la implementación del repositorio de OpenZeppelin que está disponible en `extensions/ERC20Capped.sol`, pero deberías adaptarla a las necesidades de este proyecto. Recuerda que debes controlar quién puede ejecutar esta función: en caso contrario, podrías perder el control de los *tokens* de la votación y un atacante podría manipularlos. Considera si es conveniente que sea solamente el propietario (creador) de este contrato el que pueda realizar esta operación.
- Del mismo modo, para eliminar *tokens* puedes utilizar la función interna `_burn`. Puedes mirar la implementación del repositorio de OpenZeppelin que está disponible en `extensions/ERC20Burnable.sol`. También en este caso debes controlar quién puede ejecutar esta función.

4. Detalles de implementación y pruebas

Tamaño del código compilado de los contratos. Debes tener en cuenta que existe una limitación sobre el tamaño máximo del código EVM compilado de los contratos de 24Kb aproximadamente, por lo que puede ocurrir que el contrato `QuadraticVoting` compilado supere ese tamaño máximo (el compilador muestra un mensaje de aviso si esto ocurre). En este caso, debes intentar reducir el código, por ejemplo utilizando una versión del compilador inferior a la 0.8.0 para evitar añadir comprobaciones sistemáticas de desbordamiento de expresiones enteras (y añadiendo código solo para las que deben comprobarse), o bien revisando algunas de las funciones del contrato para reducir su tamaño. Puedes hacerte una idea de cuánto debes reducir haciendo compilaciones de prueba comentando algunas funciones sencillas.

Pruebas de funcionamiento. Debes realizar pruebas de todas las funcionalidades, primero por separado, y después pruebas conjuntas. Para estas últimas puedes utilizar un guion de pruebas que ejercite el mayor número de funcionalidades posible.

Vulnerabilities. El sistema debe estar protegido frente a los ataques y vulnerabilidades vistos en clase.

5. Opcional: rediseño de `closeVoting`

En general no se recomienda implementar funciones en smart contracts que incluyan bucles sobre datos de usuario, porque pueden producirse ataques de denegación de servicio (DoS) que incrementen de tal manera los datos de usuario que se bloqueen las funciones que los procesan. En el caso de `QuadraticVoting`, la función `closeVoting` puede ser atacada de esta forma si el número de propuestas y de participantes es suficientemente alto: en ese caso, se podría bloquear el funcionamiento del sistema si no es posible ejecutar esta función en una sola transacción, ya que no sería posible cerrar la votación y devolver los *tokens* a los participantes ni ejecutar las propuestas de *signaling*.

De forma opcional, puedes rediseñar esta funcionalidad del contrato para resolver esta vulnerabilidad. Se pueden plantear dos formas de solucionar esta situación:

"Favor pull over push." Esta técnica está basada en cambiar el diseño del contrato de forma que la devolución de *tokens* y ejecución de los contratos de propuestas de *signaling* se realice bajo demanda (*pull*) en lugar de realizar estas tareas en `closeVoting` (*push*). Puedes consultar más información en:

`https://fravoll.github.io/solidity-patterns/pull_over_push.html`

`https://ethereum-contract-security-techniques-and-tips.readthedocs.io/en/latest/recommendations/#favor-pull-over-push-for-external-calls`

Si eliges esta opción, debes mantener la función `closeVoting` para que no se puedan realizar más votaciones ni añadir propuestas, y habilitar con nuevas funciones la posibilidad de que los participantes sean quienes reclamen la devolución de los *tokens* de votación y la ejecución de las propuestas de *signaling*. Debes prever un estado del contrato que no permita añadir propuestas ni votar, pero sí permita las tareas asociadas con `closeVoting`. Para ello, debes diseñar cómo debe ser el funcionamiento correcto del contrato para que todos puedan recuperar sus *tokens* después de cerrar la votación, e incluso cuando se haya reiniciado el sistema para otra votación.

"Resumable function." Otra forma de resolver este problema es plantear la función `closeVoting` como una función *reanudable* (*resumable*). Puedes ver más información sobre esta técnica en:

`https://consensys.github.io/smart-contract-best-practices/attacks/denial-of-service/#gas-limit-dos-on-a-contract-via-unbounded-operations`

La idea consiste en comprobar que se dispone de suficiente gas para realizar cada iteración del bucle, y si no es así, se guarda el número de iteración en una variable de estado para que se llame a la misma función en una transacción distinta que reanude el procesamiento del bucle desde la iteración donde se interrumpió en la transacción anterior. En el caso particular de `closeVoting` debes tener en cuenta que esta comprobación se debe realizar en varios bucles. Además, se debe evitar fijar con una constante el consumo de gas de cada iteración, salvo para la primera: debes calcular el consumo en función del coste de las iteraciones anteriores del bucle.

6. Evaluación del proyecto

La entrega debe incluir el código fuente y una memoria que describa el sistema y los detalles de diseño que consideres necesarios. Esta memoria debe incluir una sección en la que se analicen los ataques más relevantes que se pueden realizar al sistema y se describan los mecanismos que has utilizado para evitarlos.

El sistema desarrollado debe cumplir los requisitos que se han descrito en los apartados anteriores, pero además se van a valorar los siguientes aspectos:

- Debe aprovechar los recursos del lenguaje Solidity necesarios para la implementación, como por ejemplo librerías, modificadores, herencia, etc.
- Deben tenerse en cuenta los distintos tipos de vulnerabilidades para evitar posibles ataques. Ten en cuenta que este sistema de votación va a ejecutar código de contratos de propuesta externos que podrían ser maliciosos.
- También es muy importante la mantenibilidad del código, pues el contrato es complejo. No es recomendable utilizar código de bajo nivel salvo que sea imprescindible, pero se debe evitar el consumo innecesario de gas.
- Es conveniente documentar estos aspectos utilizando comentarios breves en el código fuente.
- El rediseño opcional de `closeVoting` puede suponer hasta un 15 % adicional en la puntuación del proyecto. Si lo haces, debes incluir en la memoria una sección que describa detalladamente cómo has planteado el diseño de esta parte.

La entrega del proyecto se debe realizar antes de las 23:59 del día 5 de mayo de 2024. Se deberá reservar una tutoría en el Campus Virtual la siguiente semana (6-10 de mayo) para presentar el proyecto de forma presencial y responder algunas preguntas de evaluación. Si un estudiante no asiste a la presentación y defensa de su proyecto, no será calificado.