Memoria Proyecto - **Javier Jerez Reinoso Y Pablo Agustín Chicharro Gómez**

Índice:

Estructuras internas empleadas:

Para nuestro proyecto hemos empleado un gran número de variables y estructuras de las cuales todas ellas son privadas para hacer nuestro contrato lo más seguro posible.

Las primeras variables que creamos están relacionadas con los parámetros que recibimos en el constructor del contrato:

- **address private owner:** Un address que usamos para guardar quién es el propietario del contrato, y también para evitar posibles ataques de delegateCall();

- **uint private price:** Un entero positivo para guardar el precio de cada uno de los tokens de nuestro contrato.

- **uint private maxTokens:** Otro entero para guardar el número total de tokens que se pueden comprar en nuestro contrato.

- **bool private votingOpen:** Una variable booleana que nos indica en qué momento de la votación nos encontramos, abierta o cerrada. En un principio, lo inicializamos a falso porque necesitamos que el propietario del contrato abra el proceso de votación.

- **uint private numParticipants:** Un entero que usamos como contador para llevar el número de participantes que se han registrado en nuestro contrato y que tenemos que tener para poder calcular el umbral a la hora de aprobar propuestas sin tener que emplear bucles que lo puedan hacer vulnerable.

- **Token miToken:** Como última variable privada dependiente del constructor del contrato tenemos una referencia al contrato Tokens que hemos llamado miToken, el cual es una instancia de un contrato auxiliar, que hemos implementado el cual hereda del ERC20 todas sus funcionalidades.

A parte de estas variables que las hemos puesto privadas para que, a pesar de que en la blockchain todo es accesible y público, restringir el acceso a ellas y para que solo puedan acceder los métodos de nuestro contrato, también hemos definido los siguientes mapas y estructuras:

La estructura principal del proyecto la cual se encarga de gestionar una única propuesta que hemos llamado tProposal y a partir de la cual hemos creado el resto de mapas y arrays. Esta estructura contiene los siguientes atributos:

1. **uint id:** Un identificador de la propuesta.

2. **string title:** Un título que es el identificador que nos proporciona el usuario para distinguir su propuesta.

3. **string description:** Una descripción sobre la propuesta, también proporcionada por el usuario a la hora de crear la propuesta.

4. **uint budget:** Para guardar el presupuesto asociada a la propuesta, dependiendo de su valor sabemos que tipo de propuesto tenemos, es decir, si el presupuesto es 0 entonces sabemos que va a ser de Signaling y en cualquier en otro caso(ya que estamos trabajando con uint) va a ser de Funding.

5. **address payee:** En este campo, guardamos el address del contrato que implementa la interfaz IExecutableProposal que luego en la función _checkAndExecuteProposal será utilizado.

6. **uint approved:** Esta variable se encarga de mantener el estado en el que se encuentra el contrato de entre los 5 posibles: pendiente(0), signaling(1), aprobado(2), cancelado(3) y SignalingEjecutada(4). En un principio, pensamos en utilizar una enumeración pero al final decidimos decantarnos por usar números porque usando números enteros se consume menos gas.

7. **address owner:** Guardamos la dirección del propietario de la propuesta.

8. **uint totalVotes:** Este campo lo utilizamos para llevar actualizado el número de votos que ha recibido la propuesta para así evitarnos tener que recalcular cada vez que se nos pida.

9. **uint totalTokens:** Este campo tiene la misma finalidad que el anterior pero en lugar de llevar el total de Votos, llevamos el total de tokens que tenemos.

10. **uint totalVotesAfter:** Es una variable que solo empleamos si la propuesta es de Signaling debido a la refactorización para poder mantener los votos totales que tenía dicha propuesta cuando fue cerrado su periodo de votación.

11. **uint totalTokensAfter:** Tiene el mismo funcionamiento que el parámetro anterior pero para los tokens totales

12. **uint indexIterator:** Este entero es utilizado para guardar en que posición de los arrays se encuentra la propuesta. Nos permite poder cambiar una propuesta de una lista a otra en tiempo constante evitando así el recorrer las listas.

13. **uint vPeriod:** Se encarga de guardar en qué periodo de votación se inicializó dicha propuesta.

NOTA: Cabe destacar que dentro de una estructura no es posible definir mappings en solidity y como consecuencia directa tuvimos que crear varios mapas de mapas que nos permitieran consultar tanto los votos como los tokens que tenía un participante introducido en una determinada propuesta de manera constante.

Debido a lo que hemos comentado en la **NOTA**, hemos tenido que crear dos mapas de mapas que nos permiten guardar tanto los votos como los tokens que tiene cada uno de los participantes en cada una de las propuestas. De tal forma que primero tenemos que acceder a la propuesta con su identificador y después con el address del participante podemos llegar a ver tanto los votos y los tokens sin tener que llevar a cabo una búsqueda lineal. Estos dos mapas son:

- **mapping(uint => mapping(address => uint256)) private nProposalToParticipantToNTokens;**
- **mapping(uint => mapping(address => uint256)) private nProposalToParticipantVotes;**

Además cuando una persona compra tokens(ya sea al registrarse en *addParticipant* o en la función *buyTokens*) debe de introducir Ether que es lo que hace posible que se efectúe la compra. Sin embargo, como este Ether puede no ser exacto, hemos creado un mapa que guarda el Ether sobrante para cada uno de los participantes que ha comprado Tokens en nuestro contrato. Este mapa no solo se encarga de llevar cuanto Ether sobrante tiene un participante sino que también se encarga de mantener si el participante está en el contrato o si por el contrario se ha retirado(llamando a *removeParticipant*). Para hacerlo de la manera más eficiente posible hemos decidido crear una estructura que guarde los 2 campos mencionados: el Ether restante y la condición de si el participante existe o no. Además para que ambas variables cupieran en un mismo bloque hemos utilizado uint128 en vez de uint. Esto nos asegura que el compilador va a tener espacio para guardar ambas variables en un mismo bloque, pero también implica que va ser necesario aplicar una máscara para transformar de uint a uint128 y viceversa. Esa variable se llama:

mapping(address => Participant) private participantToEther.

donde Particpant tiene la siguiente estrucura:

```
struct Participant{
    uint128 remainingEther;
    bool exists;
}
```

La opción que hemos optado para la gestión del Ether restante es que si un mismo participante quiere comprar más tokens tiene el Ether restante disponible y cuando un participante decide retirarse del contrato, ese ether sobrante se mantendrá registrado en el

contrato por si decide volver. Sin embargo, si ese participante quiere vender todos sus Tokens pero no quiere eliminarse del contrato, entonces ahí sí que se le devolvería todo el Ether restante junto con el precio de todos los tokens que hubiera comprado.

Para poder tener controlado en qué estado se encuentra cada propuesta, y además poder ejecutar las funciones que nos piden devolver listas con las propuestas en un tiempo lineal pero controlado, hemos definido cuatro arrays que son los siguientes:

- **uint[] private proposalPending**
- **uint[] private proposalCancelled**
- **uint[] private proposalSignaling**
- **uint[] private proposalApproved**

Hemos optado por esta opción en vez de tener un array con todas las propuestas de todos los tipos de estados ya que así no tenemos que recorrer todas las propuestas cada vez que pidamos una de las listas y tenemos todo construido en storage. Además gracias al recurso de llevar el índice en el que se encuentra cada propuesta, podemos borrar y añadir elementos de manera eficiente sin tener que recorrer estos mapas.

Necesitamos también una variable **bool private lock** para poder proteger nuestro contrato de ataques de Reentrada.

Finalmente, como último atributo privado llevamos un mapa de enteros a tProposal:

mapping(uint => tProposal) private idToProposal

que contiene todas las propuestas que se han introducido en el contrato sin importar su estado para poder llevar a cabo consultas de manera eficiente como por ejemplo el número de votos totales de una propuesta u . Para que este mapa funcione tenemos un índice interno (*nProposal*) que nos sirve para poder diferenciar una propuesta de otra que se irá incrementando cuando se añadan propuestas en *addParticipant*.

Emits:

Hemos definido algunos eventos en el contrato que indican estados de ejecución.

- **event Print(**string message**):** Para poder imprimir diferentes mensajes.

- **event TokensPurchased(**address buyer, uint256 amount**):** Para que nos devuelva quién y cuántos tokens se han comprado.

- **event TokensSold(**address buyer, uint256 amount**):** Para que nos devuelva quién y cuántos tokens se han vendido.

- **event ProposalApprovedMSG(**uint id**):** Para saber qué propuesta se ha aprobado.

- **event ProposalCancelledMSG(**uint id**):** Para saber qué propuesta se ha cancelado.

Modifiers:

- **OnlyAuthority():** Compara si el **msg.sender** es igual que el **owner** del contrato (definido previamente en el constructor).

- **participantExists(**address p**):** Comprueba que el valor del **address p** proporcionado no sea igual a **uint(0)**, es decir, comprueba que ese participante exista.

- **participantNotExists(**address p**):** Comprueba si existe el participante con el atributo *exists* del mapa de participantes.

- **onlyCreator(**address a, address b**):** Comprueba si ambos addresses son iguales.

- **enoughCredit(**uint value**):** Comprueba si el valor introducido por el usuario es mayor que el precio del token.

- **onlyOpened():** Comprueba si la votación está abierta.

- **stillTokensLeft():** Comprueba si el número actual de tokens comprados es menor que el número máximo de tokens disponibles.

- **notApprovedProposal(**uint pId**):** Comprueba si la propuesta ha sido aprobada.

- **SignalingProposal(**uint pId**):** Comprueba si la propuesta es de tipo signaling.

- **samePeriod(**uint period**):** Comprueba si el periodo de la votación es el mismo que el del parámetro introducido.

- **notSamePeriod(**uint period**):** Comprueba si el periodo de la votación es diferente que el del parámetro introducido.

Métodos auxiliares:

- **calcularUmbral(**uint budget**):** Calculamos el umbral con la fórmula. Como Solidity trabaja con números enteros positivos, calcularemos el divisor común que sería $5 \cdot totalBudget$, ya que 0.2 es equivalente a $\frac{1}{5}$. El objetivo de la función es respetar el cálculo matemático en vez de los paréntesis

  De esta forma, pasamos de la siguiente fórmula:

  $$threshold_i = (0.2 + \frac{budget_i}{totalBudget}) \cdot numParticipants + numPendingProposals$$

  A la siguiente fórmula:

  $$threshold_i = \frac{(totalBudget + 5 \cdot budget_i) \cdot numParticipants}{5 \cdot totalBudget} + numPendingProposals$$

- **evaluarProposal(**uint pid**):** En esta función comprobamos si una propuesta de tipo funding es aprobada. Para que una propuesta de tipo funding se apruebe se tienen que dar las siguientes condiciones:

  1. Primero comprobamos que el presupuesto total del proceso de votación es suficiente en el momento de evaluar el umbral.
  2. El número de votos recibidos en esa propuesta supera el umbral.

- **powerOf2(**uint x**):** Es una función private y pure, ya que no cambia el estado del contrato y tampoco accede a variables de este. Nos permite calcular el cuadrado de un número.

- **executeSignaling(**int id**):** Es una función public. Para poder ser ejecutada debe ser llamada por el propietario de la propuesta. La propuesta introducida sólo podrá ser de tipo Signaling y tendrá que ser llamada en un periodo de votación diferente al actual. Llamamos a la función externa de executeProposal() con los tokens que tenía nuestra propuesta en el momento en el que fue aprobada y cambiamos el estado de la propuesta de Signaling a Signaling Ejecutada para que el propietario no la pueda ejecutar más de una vez.

Métodos primarios del contrato:

- **openVoting():** Es una función public y payable; sin embargo, solo el owner del contrato puede ejecutar esta función. Inicializa el budget total del contrato, empieza el proceso de votación e incrementa el periodo de votación.

- **addParticipant():** Es una función public y payable. Solamente se puede llamar a la función si hay crédito suficiente para comprar como mínimo un token. Si introduce más crédito que el del precio de un token, se comprarán todos los tokens posibles con la cantidad introducida y el dinero sobrante se guardará en la base de datos del

contrato. Finalmente se incrementará el número de participantes del contrato y se le otorgarán los tokens a ese usuario en el ERC20.

- **removeParticipant():** Es una función public y payable. Después de comprobar que el participante existe. Reduce el número de participantes y pone la booleana que indica si el participante existe a **false**. (NO borramos la información del participante del sistema, solamente lo marcamos como no existente).

- **addProposal(**string memory title_, string memory description_, uint budget_, address payee_**):** Es una función public. Comprueba el participante existe y si el proceso de votación está abierta. Crea una estructura proposal y la inicializa. Añade la nueva propuesta al mapa de propuestas, y dependiendo de si tiene budget o no, se añadirá a la lista de Signaling o a la de Pending proposals.

- **cancelProposal(**uint id**):** Es una función public. Es necesario que la votación esté abierta, que esté en el mismo periodo de votación y que el que ejecute la función sea el propietario de la propuesta. Si la propuesta no ha sido aprobada, la quitamos del array de propuestas de pending o de signaling, la marcamos como cancelada y la añadimos a la lista de propuestas canceladas.

- **buyTokens():** Es una función public y payable. Primero de todo se comprueba si el participante existe, si introduce crédito suficiente para comprar al menos un token y si hay tokens disponibles en stock. En este caso, usaremos también el dinero sobrante que tiene el participante almacenado de otras compras para realizar el pago. Una vez más, el dinero sobrante será almacenado y los tokens comprados serán incluidos al ERC20 del participante.

- **sellTokens(**uint sell**):** Es una función public y payable. Se comprueba que el participante exista y que la cantidad de tokens que quiere vender es menor o igual a la cantidad de tokens que tiene (para evitar problemas de overflow). Devolvemos el equivalente en dinero del número de tokens que quiere vender junto con el dinero sobrante que tiene guardado ese usuario. Quemamos el número de tokens del participante en el ERC20.

- **getERC20():** Es una función que es public y view, ya que solamente vamos a consultar su contenido. Nos devuelve la dirección del ERC20 interno que tenemos en nuestro contrato, es decir, miToken.

- **getPendingProposals():** Es una función public y view, la cual se encarga de recuperar la lista interna que tenemos de propuestas pendientes. Solo se puede ejecutar si la propuesta está abierta.

- **getApprovedProposals():** Es una función public y view, la cual se encarga de retornar la lista interna que tenemos de propuestas aprobadas. Solo se puede ejecutar si la propuesta está abierta.

- **getSignalingProposals():** Es una función public y view, la cual se encarga de recuperar la lista interna que tenemos de propuestas que son de tipo Signaling. Solo se puede ejecutar si la propuesta está abierta.

- **getProposalInfo(**uint id**):** Es una función public y view (ya que solo vamos a leer información). Retorna una descripción con toda la información de la proposal.

- **stake(**uint votes, uint id**):** Es una función public. Lo primero que hacemos es comprobar que el periodo de votación se encuentra abierto y que además el participante que quiere ejecutar la función se ha registrado en nuestro contrato, además de que esté en el mismo periodo de votación. Además, comprobamos si esta función está bloqueada con el *lock* o no, para evitar posibles vulnerabilidades.

  Una vez ya hemos comprobado todas las condiciones previas, lo primero que debemos hacer es ver cuantas veces ya ha votado ese participante, debido a que cada voto necesita de un número diferente de tokens por emplear el sistema de votación quadraticVoting. Además, necesitamos comprobar que externamente el participante ha cedido ese número de tokens a nuestro contrato, lo cual se hace en el propio *transferFrom* con el *_spendAllowance*. A continuación, si todo ha ido correctamente, es el momento en el que actualizamos todos los mapas con los votos y tokens añadidos. Finalmente, si la propuesta es de tipo Funding, comprobamos gracias a la función interna _checkAndExecuteProposal si ya se han cumplido los requisitos para aprobar dicha propuesta.

- **withdrawFromProposal(**uint votes, uint id**):** Es una función public. Antes de nada comprobamos que la propuesta no ha sido aprobada y que el participante existe. También nos aseguramos si esta función está bloqueada con el *lock* o no, para evitar posibles vulnerabilidades. Comprobamos también que el número de votos que quiere retirar es válido (para evitar problemas de underflow). Calculamos el número de tokens a devolver al usuario y los introducimos en el ERC20 del participante. Finalmente actualizamos los mapas y la información de la proposal con el número de votos y de tokens resultante.

- **_checkAndExecuteProposal(**uint id**):** Es una función que es internal y no tiene ningún modificador porque queremos que se ejecute siempre aún si no se puede aprobar la propuesta que tiene que comprobar. A pesar de no tener modificadores sí que necesitamos hacer comprobaciones ya que esta función sólo tendrá efecto si su evaluación es satisfactoria, algo que hacemos llamando a la función privada *evaluarProposal* y no esté bloqueada por el *lock*, para evitar vulnerabilidades.

  Si la propuesta es aprobable debemos de hacer una llamada externa (protegiendo la llamada) al contrato que implementa la interfaz *IExecutableProposal* que tenemos guardado en la propuesta, transfiriendole el dinero que tenemos almacenado en budget. Además, debemos de actualizar el presupuesto total del contrato y finalmente debemos de eliminar la propuesta de la lista de propuestas pendientes para añadirlas a la lista de propuestas aprobadas, utilizando el indexIterator de la propuesta para hacerlo en tiempo constante y no tener que recorrer las listas. También se actualiza el estado de la propuesta en su estructura.

CLOSE VOTING SIN IMPLEMENTAR LA PARTE OPCIONAL

- **closeVoting():** Es una función public y payable. Solamente la puede ejecutar el propietario del contrato. Esta función cierra el proceso de votación. Además, es la función más cara del contrato debido a que recorre todas las listas. Este gran consumo de gas lo resolveremos con el **favour pull over push** de la parte opcional.

  Primero recorre las listas de pendingProposals, SignalingProposals y cancelledProposals para devolver los tokens a los usuarios que han votado en esas propuestas. Para ello usaremos la función auxiliar returnTokens(). Finalmente, el presupuesto sobrante del contrato se transfiere al propietario.

CLOSE VOTING IMPLEMENTANDO LA PARTE OPCIONAL

- **closeVoting():** Es una función public y payable. Solamente la puede ejecutar el propietario del contrato mientras el proceso de votación esté abierto. En ella solamente ponemos la variable del proceso de votación a **false** y transferimos el presupuesto total del contrato al propietario.

Concepto de la propiedad:

En nuestro contrato, el concepto de propiedad de Tokens es algo muy importante y que además es muy necesario a la hora de implementar correctamente todas las funcionalidades requeridas.

En nuestro contrato, nos hemos asegurado que si una dirección/participante no es propietario o está autorizado a usar unos determinados Tokens esta persona no va a poder llevar a cabo la transacción. Esto se debe a que una dirección externamente a nuestro contrato puede hacer lo que quiera con los tokens que son suyos, y nosotros en el Quadratic voting lo único que registramos es cuando una persona nos cede sus tokens para una propuesta.

Es por ello, que no hemos definido ningún mapa que contenga el número de Tokens que tiene en total un usuario, porque además lo único que debemos saber es cuántos tokens tiene cada una de las propuestas de nuestro contrato.

Vulnerabilidades:

- **Underflow/Overflow:**

La primera vulnerabilidad a la que nos hemos enfrentado es la de Overflow, ya que al elegir una versión del compilador superior o igual a la 0.8 el propio compilador se encarga de llevar a cabo todas las posibles comprobaciones y por lo tanto evita que se produzcan esos ataques. Para el caso de los underflows, también hemos hecho las comprobaciones necesarias con **require's** al principio de las funciones para no obtener resultados negativos de las operaciones.

Sin embargo, el comprobar el overflow y underflow gracias a tener la versión 0.8 del compilador añade coste de gas a nuestro contrato y es por eso que se puede emplear la función unchecked en determinados puntos (operaciones aritméticas) para no comprobar dicho overflow/underflow si el programador está seguro de que no se pueden producir.

Hemos decidido que no es necesario comprobar el overflow y underflow en tres puntos de nuestro contrato debido a que nunca se puede producir:

→ En la variable uint nProposals que lleva el total de propuestas que se han registrado en nuestro contrato. Esto se debe a que nunca va a poder ser un número negativo porque el tipo de la propia variable lo impide y tampoco se puede dar el caso de desbordar la variable porque no es realista que se registren 2^{256} propuestas.

→ Lo mismo ocurre con la variable nParticipants ya que cumple exactamente las mismas condiciones que en el caso anterior.

→ Dentro de la función withdrawFromProposal(), cuando nosotros calculamos la variable tokenReturned. El no tener que añadir comprobaciones de overflow ni underflow se debe a que primero con el require de la línea anterior nos aseguramos que nunca vamos a tener un número negativo ya que como mínimo tendremos 0 como resultado y tampoco vamos a tener overflow ya que a pesar de que vamos a elevar al cuadrado las variables, el número de votos se ve influenciado directamente por el número de tokens totales que permite tener nuestro contrato en la variable uint maxTokens, que nunca va a tener valores cercanos a 2^{256} lo que hace que esta operación no sea necesaria de proteger.

El máximo de tokens posibles sería 2^{128} ya que:

$$(2^{128})^2 = 2^{128 \cdot 2} = 2^{256}$$

y ese valor no es realista.

- **Reentrancy attack:**

Para el ataque de reentrada con el cual el atacante nos podría llegar a robar ether y tokens al llamar a otro contrato que llama a una función que no existe en nuestro contrato accediendo así al fallback. Es por ello que debemos de proteger las llamadas externas

utilizando locks lo cual nos permite que hasta que todas las variables no se actualizaron, no se puede continuar con la ejecución y bloquea el poder hacer el ataque de reentrada. En particular en nuestro contrato, este posible ataque se puede dar en la función **checkAndExecuteProposal, stake, executeSignaling y withdrawFromProposal** ya que es el momento en el que hacemos la llamada externa al contrato que implementa la interfaz iExecuteProposal y lo hemos protegido con los *locks*.

- **DelegateCall: (Parity Wallet)**

En nuestro contrato no tenemos problemas de Parity Wallet debido a que no hacemos uso de una librería para establecer el propietario del contrato. Además, inicializamos el owner del contrato en el constructor en vez de hacerlo con una llamada externa usando delegateCall a la librería, por lo que nunca tendremos este problema en nuestro contrato.

- **tx.origin authentication:**

En nuestro contrato siempre usamos **msg.sender** en vez de **tx.origin** para la autorización del propietario porque **msg.sender** refleja la dirección del contrato que está llamando al método, lo cual es más seguro y previene ataques de reentrancy. En cambio, **tx.origin** devuelve la dirección del remitente original de la transacción, que podría ser diferente si se llama a través de otro contrato, permitiendo así alguna vulnerabilidad.

- **Block Timestamp Manipulation:**

Dado que no hacemos uso de timestamps nuestro contrato no corre riesgo frente a este tipo de vulnerabilidad.

• <u>PARTE OPCIONAL:</u>

Favour pull over push:

Al implementar el *favour pull over push* nos protegemos de un posible Denial of Service (DoS). El DoS se puede dar en el caso de que, en la versión sin esta parte opcional, tengamos un número elevado de propuestas o un gran número de participantes en una misma propuesta, de manera que tengamos que recorrer unos bucles muy grandes en la función **closeVoting()** y nos podamos quedar sin gas para ejecutar el contrato.

El proceso de transformación de la parte original a la parte opcional ha sido el siguiente.

Primero de todo, hemos incluido un período de votación general y otro particular en cada propuesta, de manera que los participantes solamente podrán votar en una propuesta si el periodo de votación del contrato y el de esa propuesta coinciden, aparte de que el proceso de votación esté abierto.

Por otra parte, una propuesta podrá ser cancelada en cualquier periodo de votación. De esta forma, un participante podrá retirar votos de una propuesta y recuperar sus tokens cuando una propuesta está pending, signaling o cancelled. Para ello, hemos usado la función **withdrawFromProposal** del contrato original y le hemos quitado el modifier **cancelledProposal(**uint pId**)** para que pueda ser llamada también con propuestas canceladas.

Además, el periodo de votación general se incrementa cada vez que comienza una nueva votación. Con todos estos cambios, permitimos a los participantes recuperar sus tokens de las propuestas que no han sido aprobadas, pudiendo quitar todos los bucles de la función **closeVoting()**, y por tanto, del contrato.

Por último, las propuestas de tipo Signaling han de ejecutarse al terminar el proceso de votación. Para mantener el funcionamiento del *favour pull over push*, hemos creado la función executeSignaling, que permite al propietario de esa propuesta Signaling ejecutarla una vez acabado el proceso de votación (en un periodo de votación diferente), gracias a las dos variables que contabilizan el número de tokens y de votos en el momento que la propuesta es aprobada.

Con esta implementación de la parte opcional, hemos conseguido un contrato final que no contenga ningún bucle.

CASO DE USO:

Creamos el contrato con el precio de cada token a 10 weis y el máximo de tokens a 50.
Abrimos el contrato con un presupuesto total de 8000 weis.

Añadimos al participante [1] con 5 tokens, que tiene address 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2

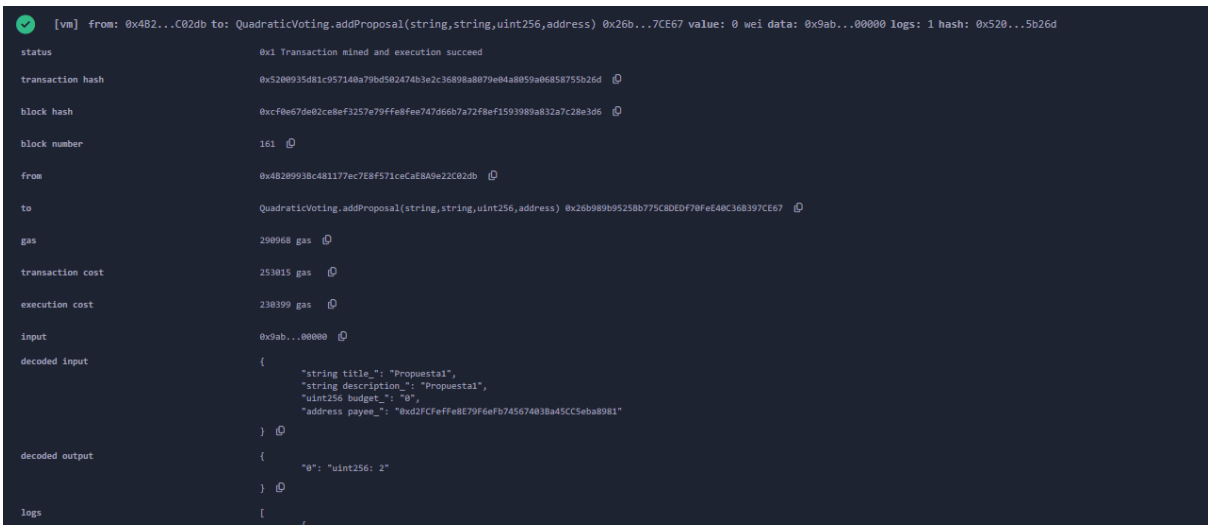
Añadimos al participante [2] con 16 tokens, que tiene address 0x4B20993Bc481177ec7E8f571ceCaE8A9e22C02db

Añadimos al participante [3] con 24 tokens, que tiene address 0x78731D3Ca6b7E34aC0F824c42a7cC18A495cabaB

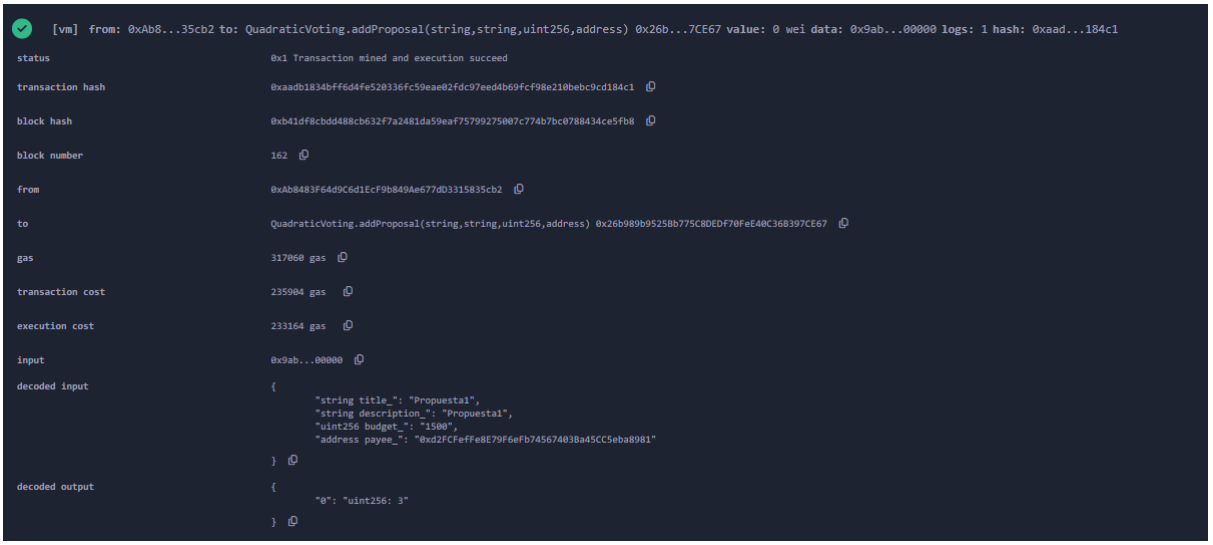
Aprobamos todos los tokens de los participantes con el approve. Después corremos el ERC20 con el address del getERC20 del contrato.

A continuación, creamos una propuesta tipo pending con un budget de 2000 weis y el address del contrato auxiliar como payee_

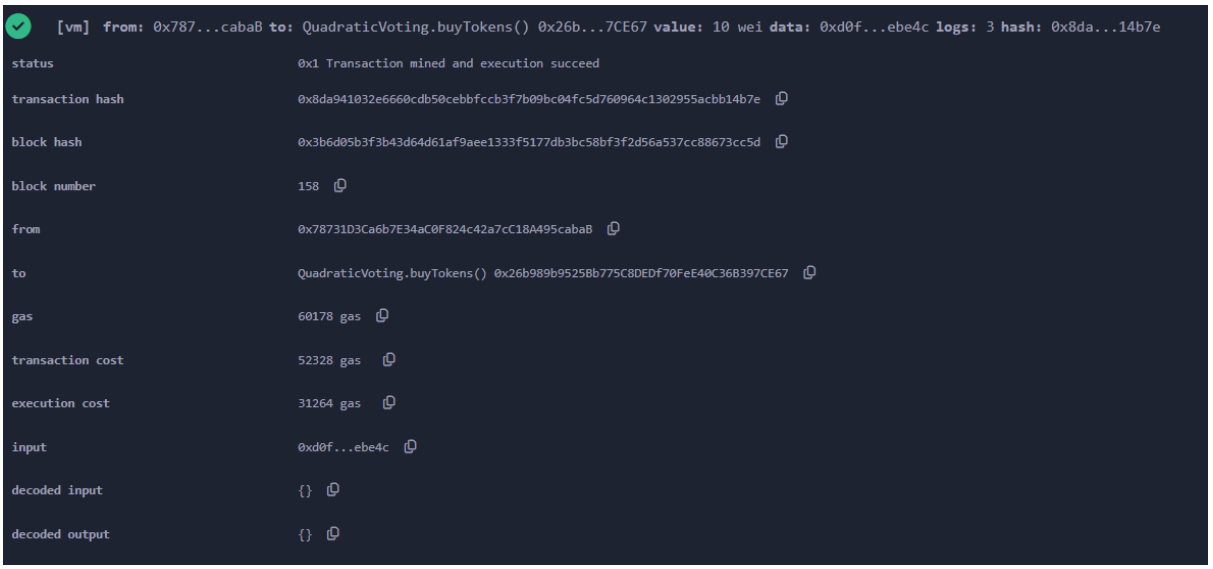
De la misma manera creamos la proposal de tipo Signaling pero con un budget de 0.



Creamos una tercera proposal, está será de tipo pending con un budget de 1500 weis.



Compramos un token con el participante [3], ahora tiene 25 tokens



```
[vm] from: 0x787...cabaB to: QuadraticVoting.buyTokens() 0x26b...7CE67 value: 10 wei data: 0xd0f...ebe4c logs: 3 hash: 0x8da...14b7e

status                  0x1 Transaction mined and execution succeed
transaction hash        0x8da941032e6660cdb50cebbfccb3f7b09bc04fc5d760964c1302955acbb14b7e
block hash              0x3b6d05b3f3b43d64d61af9aee1333f5177db3bc58bf3f2d56a537cc88673cc5d
block number            158
from                    0x78731D3Ca6b7E34aC0F824c42a7cC18A495cabaB
to                      QuadraticVoting.buyTokens() 0x26b989b9525Bb775C8DEDf70FeE40C36B397CE67
gas                     60178 gas
transaction cost        52328 gas
execution cost          31264 gas
input                   0xd0f...ebe4c
decoded input           {}
decoded output          {}
```


```
logs [
    {
        "from": "0x5886B2693f246fc2171d1a982CB4F7C1A4D25223",
        "topic": "0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef",
        "event": "Transfer",
        "args": {
            "0": "0x0000000000000000000000000000000000000000",
            "1": "0x78731D3Ca6b7E34aC0F824c42a7cC18A495cabaB",
            "2": "1",
            "from": "0x0000000000000000000000000000000000000000",
            "to": "0x78731D3Ca6b7E34aC0F824c42a7cC18A495cabaB",
            "value": "1"
        }
    },
    {
        "from": "0x26b989b9525Bb775C8DEDf70FeE40C36B397CE67",
        "topic": "0x241ba3bafc919fb4308284ce03a8f4867a8ec2f0401445d3cf41a468e7db4ae0",
        "event": "Print",
        "args": {
            "0": "Se han comprado Tokens",
            "message": "Se han comprado Tokens"
        }
    },
    {
        "from": "0x26b989b9525Bb775C8DEDf70FeE40C36B397CE67",
        "topic": "0x8f28852646c20cc973d3a8218f7eefed58c25c909f78f0265af4818c3d4dc271",
        "event": "TokensPurchased",
        "args": {
            "0": "0x78731D3Ca6b7E34aC0F824c42a7cC18A495cabaB",
            "1": "1",
            "buyer": "0x78731D3Ca6b7E34aC0F824c42a7cC18A495cabaB",
            "amount": "1"
        }
    }
]
```

Vendemos un token del participante [1], ahora tiene 4 tokens

```
[vm] from: 0xAb8...35cb2 to: QuadraticVoting.sellTokens(uint256) 0x26b...7CE67 value: 0 wei data: 0x6c1...00001 logs: 3 hash: 0x1cb...47201

status                      0x1 Transaction mined and execution succeed
transaction hash            0x1cb85d9fd5e6cc647b5146d961bcb5d518b43f29b840aeb8711f68ca74547201
block hash                  0x02be695d5ec262ae1d6d9743a4e4ae0bb820d4b0a992dc1ea7a1525692af9f4e
block number                166
from                        0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2
to                          QuadraticVoting.sellTokens(uint256) 0x26b989b9525Bb775C8DEDf70FeE40C36B397CE67
gas                         65989 gas
transaction cost            57381 gas
execution cost              36177 gas
input                       0x6c1...00001
decoded input               {
                                "uint256 sell": "1"
                            }
decoded output              {}
```

```
logs                        [
                                {
                                    "from": "0x5886B2693f246fc2171d1a982CB4F7C1A4D25223",
                                    "topic": "0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef",
                                    "event": "Transfer",
                                    "args": {
                                        "0": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2",
                                        "1": "0x0000000000000000000000000000000000000000",
                                        "2": "1",
                                        "from": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2",
                                        "to": "0x0000000000000000000000000000000000000000",
                                        "value": "1"
                                    }
                                },
                                {
                                    "from": "0x26b989b9525Bb775C8DEDf70FeE40C36B397CE67",
                                    "topic": "0x241ba3bafc919fb4308284ce03a8f4867a8ec2f0401445d3cf41a468e7db4ae0",
                                    "event": "Print",
                                    "args": {
                                        "0": "Se han vendido Tokens",
                                        "message": "Se han vendido Tokens"
                                    }
                                },
                                {
                                    "from": "0x26b989b9525Bb775C8DEDf70FeE40C36B397CE67",
                                    "topic": "0x57d61f3ccd4ccd25ec5d234d6049553a586fac134c85c98d0b0d9d5724f4e43e",
                                    "event": "TokensSold",
                                    "args": {
                                        "0": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2",
                                        "1": "1",
                                        "buyer": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2",
                                        "amount": "1"
                                    }
                                }
```

Algunas de las votaciones:

```
[vm] from: 0xAb8...35cb2 to: QuadraticVoting.stake(uint256,uint256) 0x26b...7CE67 value: 0 wei data: 0x7b0...00001 logs: 1 hash: 0xec5...8a379

console.log:
ESTAMOS DENTRO DEL STAKE
1
DENTRO DEL IF DEL CHECK
ESTAMOS DENTRO DEL CHECK
EL THRESHOLD ES:
3
```

```
[vm] from: 0xAb8...35cb2 to: QuadraticVoting.stake(uint256,uint256) 0x26b...7CE67 value: 0 wei data: 0x7b0...00001 logs: 1 hash: 0x861...4f404
console.log:
ESTAMOS DENTRO DEL STAKE
3
DENTRO DEL IF DEL CHECK
ESTAMOS DENTRO DEL CHECK
EL THRESHOLD ES:
3
```

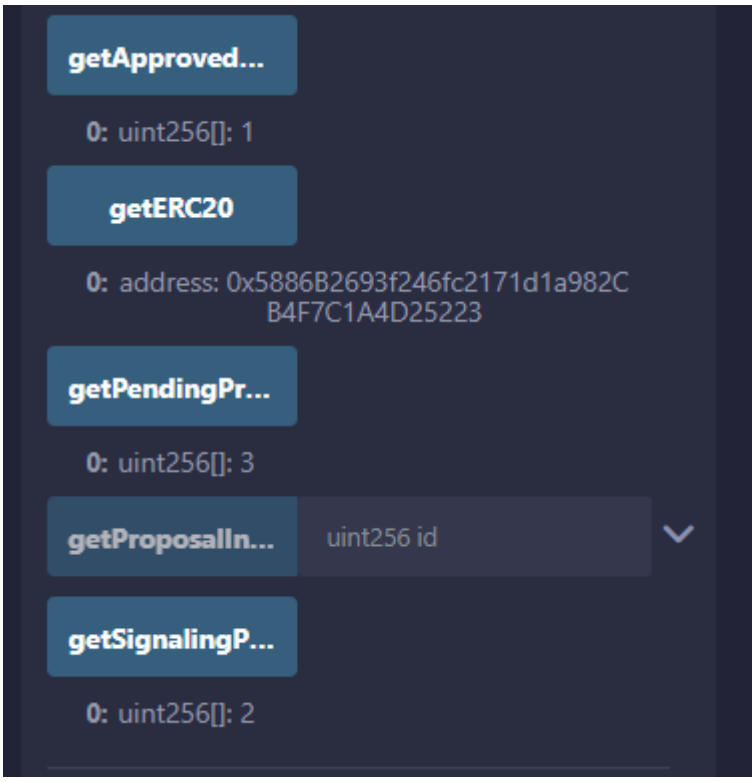
Este intento de propuesta da un error porque el participante se ha quedado sin tokens.

```
[vm] from: 0xAb8...35cb2 to: QuadraticVoting.stake(uint256,uint256) 0x26b...7CE67 value: 0 wei data: 0x7b0...00001 logs: 0 hash: 0xbaf...faf6d
console.log:
ESTAMOS DENTRO DEL STAKE
5
transact to QuadraticVoting.stake errored: Error occurred: revert.

revert
        The transaction has been reverted to the initial state.
Error provided by the contract:
ERC20InsufficientBalance
Parameters:
{
 "sender": {
  "value": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2",
  "documentation": "Address whose tokens are being transferred."
 },
 "balance": {
  "value": "0",
  "documentation": "Current balance for the interacting account."
 },
 "needed": {
  "value": "5",
  "documentation": "Minimum amount required to perform a transfer."
 }
}
You may want to cautiously increase the gas limit if the transaction went out of gas.
```

Podemos comprobar que la propuesta se aprueba con la segunda votación, y con la librería de hardhat podemos ver los resultados y valores de las variables durante la ejecución.

```
[vm] from: 0x787...cabaB to: QuadraticVoting.stake(uint256,uint256) 0x26b...7CE67 value: 0 wei data: 0x7b0...00001 logs: 1 hash: 0x731...eeac8
console.log:
ESTAMOS DENTRO DEL STAKE
1
DENTRO DEL IF DEL CHECK
ESTAMOS DENTRO DEL CHECK
EL THRESHOLD ES:
3
transact to QuadraticVoting.stake pending ...

[vm] from: 0x787...cabaB to: QuadraticVoting.stake(uint256,uint256) 0x26b...7CE67 value: 0 wei data: 0x7b0...00001 logs: 4 hash: 0xd20...a23ac
console.log:
ESTAMOS DENTRO DEL STAKE
3
DENTRO DEL IF DEL CHECK
ESTAMOS DENTRO DEL CHECK
EL THRESHOLD ES:
3
ESTAMOS DENTRO DEL IF DEL CHECK
NUMERO DE VOTOS DE LA PROPUESTA:
4
NUMERO DE TOKENS DE LA PROPUESTA:
8
HEMOS HECHO LA LLAMADA EXTERNA
EL TOTAL BUDGET ES:
6080
```

Como podemos comprobar, la proposal se añade correctamente a la lista de aprobadas.

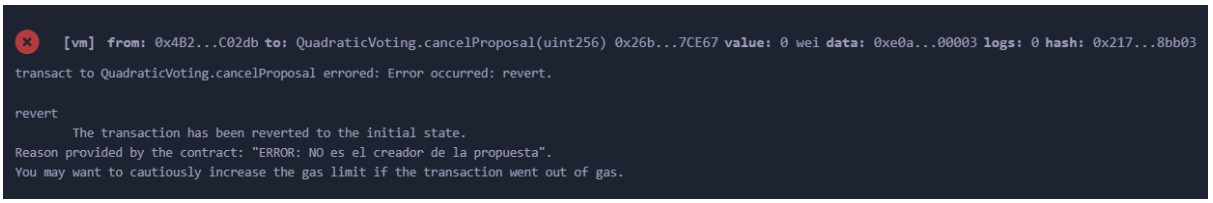


Realizamos un withdraw from proposal en el mismo periodo de votación con el proceso de votación abierto.

```
[vm] from: 0x787...cabaB to: QuadraticVoting.withdrawFromProposal(uint256,uint256) 0x26b...7CE67 value: 0 wei data: 0x649...00002 logs: 3 hash: 0x270...19cd9

status                  0x1 Transaction mined and execution succeed
transaction hash        0x270a982ad24c81cd8bab04f2fa0dda2a595ec467dca3d1a8856c7664c9e19cd9
block hash              0xad3d8fc1670cc39339533531d94f93a7dfc66657802a1936f7e9f9466def5297
block number            174
from                    0x78731D3Ca6b7E34aC0F824c42a7cC18A495cabaB
to                      QuadraticVoting.withdrawFromProposal(uint256,uint256) 0x26b989b9525Bb775C8DEDf70FeE40C36B397CE67
gas                     96174 gas
transaction cost        83629 gas
execution cost          62285 gas
input                   0x649...00002
decoded input           {
                            "uint256 votes": "2",
                            "uint256 id": "2"
                        }
decoded output          {}
```

```
logs                    [
                            {
                                "from": "0x5886B2693f246fc2171d1a982CB4F7C1A4D25223",
                                "topic": "0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef",
                                "event": "Transfer",
                                "args": {
                                    "0": "0x26b989b9525Bb775C8DEDf70FeE40C36B397CE67",
                                    "1": "0x78731D3Ca6b7E34aC0F824c42a7cC18A495cabaB",
                                    "2": "8",
                                    "from": "0x26b989b9525Bb775C8DEDf70FeE40C36B397CE67",
                                    "to": "0x78731D3Ca6b7E34aC0F824c42a7cC18A495cabaB",
                                    "value": "8"
                                }
                            },
                            {
                                "from": "0x26b989b9525Bb775C8DEDf70FeE40C36B397CE67",
                                "topic": "0x241ba3bafc919fb4308284ce03a8f4867a8ec2f0401445d3cf41a468e7db4ae0",
                                "event": "Print",
                                "args": {
                                    "0": "Se han devuelto Tokens",
                                    "message": "Se han devuelto Tokens"
                                }
                            },
                            {
                                "from": "0x26b989b9525Bb775C8DEDf70FeE40C36B397CE67",
                                "topic": "0x57d61f3ccd4ccd25ec5d234d6049553a586fac134c85c98d0b0d9d5724f4e43e",
                                "event": "TokensSold",
                                "args": {
                                    "0": "0x78731D3Ca6b7E34aC0F824c42a7cC18A495cabaB",
                                    "1": "8",
                                    "buyer": "0x78731D3Ca6b7E34aC0F824c42a7cC18A495cabaB",
                                    "amount": "8"
                                }
                            }
                        ]
```

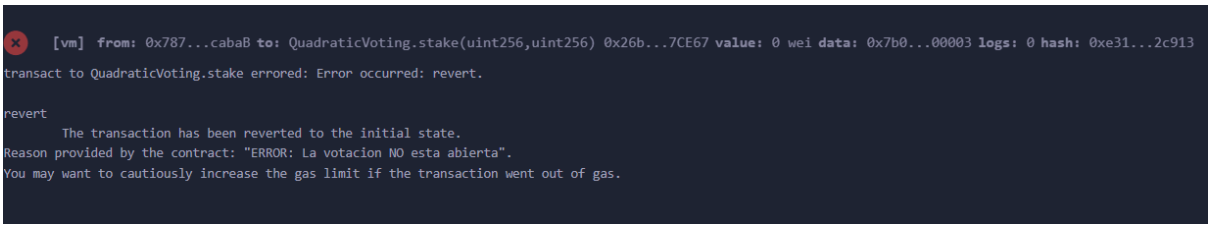
En este caso salta un error debido a que intentamos cancelar una propuesta desde un participante que no es el propietario de esa propuesta.



```
[vm] from: 0x4B2...C02db to: QuadraticVoting.cancelProposal(uint256) 0x26b...7CE67 value: 0 wei data: 0xe0a...00003 logs: 0 hash: 0x217...8bb03
transact to QuadraticVoting.cancelProposal errored: Error occurred: revert.

revert
        The transaction has been reverted to the initial state.
Reason provided by the contract: "ERROR: NO es el creador de la propuesta".
You may want to cautiously increase the gas limit if the transaction went out of gas.
```

Ahora que hemos puesto el address correcto, cerramos la propuesta correctamente.

Si ahora intentamos votar en la propuesta cancelada, saltará un error.



Una vez el proceso de votación haya sido cerrado, la propuesta de signaling podrá ser ejecutada por su propietario.

Cabe destacar que antes de ejecutar la propuesta signaling hemos quitado un voto del participante [3]. Sin embargo, podemos comprobar que el número de votos es 3 debido a que ese es el número de votos de la propuesta en el momento de aprobarse.