Proyecto final

Javier Mulero Martín y Jorge del Valle Vázquez

19 de mayo de 2022

1. Introducción

El objetivo de este proyecto es la implementación de un **sistema de votación cuadrático** de propuestas *on chain* para DAOs.

Las DAOs son **organizaciones autónomas descentralizadas** que se definen por unas **reglas públicas e inmodificable** es de gobernanza guardadas en la **Blockchain**.

Una de las formas más comunes de gobernanza en una DAO es por votación y una opción sencilla es la que utiliza la **votación cuadrática**. Esta votación consiste en incrementar el coste de los votos de un miembro de manera exponencial. Hay que tener en cuenta, que si un participante da dos votos a una propuesta (le cuesta 4 tokens) y quiere volver a dar otro, el siguiente le costará $9 - 4 = 5$ tokens, ya que es el precio que le falta para mantener el coste cuadrático con 3 votos.

Número de votos	Tokens
1	1
2	4
3	9
4	16
...	...

Cuadro 1: Votación cuadrática por medio de tokens

Para implementar este sistema de votación, se van a utilizar tokens ERC20, de forma que cada voto cuesta un número determinado de tokens en función del coste cuadrático.

Vamos a seguir las implementaciones de OpenZeppelin:

- ERC20.sol, que implementa la interfaz IERC20.sol, IERC20Metadata.sol y hereda de Context.sol.
- ERC20Capped.sol.
- ERC20Burnable.sol.

Observamos que en nuestro contrato `ERC20TC`, el `_controller` será el único que pueda hacer `mint` y `burn` de los tokens, y en nuestro casó serña el contrato `QuadraticVoting`, pues es quien crea el contrato `ERC20TC`.

2. Implementación de las funciones de QuadraticVoting

Vamos a comentar primero la estructura de cada *proposal*, algunos atributos del contrato, y después comentaremos los detalles de cada función implementada.

De cada *proposal* almacenamos su título, descripción, budget, dirección del contrato, el creador de la propuesta, si ha sido aprobada o cancelada el número de votos y correspondientes tokens que ha recibido.

De entre las estructuras empleadas destacamos `owner` (creador del contrato QV), `tokenPrice` (precio de los tokens), `totalBudget` (presupuesto total disponible), y dos mappings, `participantVotes` que registra los para cada participante el número de votos que dedica a cada propuesta, y `proposals` que registra las proposals indexadas por un identificador entero, cada uno con su correspondiente iterador.

Se emplean múltiples *modifiers* que comprueban si es el owner, si es un nuevo participante o ya lo era, si QV tiene tokens para transferir, si el participante aporta lo suficiente para comprar un toen, si la votación está abierta, si se es el creador de una proposal, si se trata de una propuesta añadida y si la propuesta no ha sido cancelada ni aprobada de momento.

Ahora pasamos a comentar cada función.

2.1. constructor

- Se guarda el `owner` como `payable` para transferirle el presupuesto sobrante al cerrar la votación.
- Almacenamos el valor de los tokens en `tokenPrice`.
- Creamos el contrato ERC20 y creamos con `mint` hasta el máximo establecido por `_supply`.
- Se indica que la votación no está iniciada y se da 1 al primer id que pueda tener una propuesta que irá incrementándose.

2.2. openVoting

- Inicia la votación comprobando que no haya una en curso.
- Establece el presupuesto inicial que el creador de QV decida aportar.
- Se crean tokens hasta el máximo permitido, con vistas a recuperar los tokens quemados con `burn` por la ejecución de proposals en la votación anterior (en la primera

apertura sería de 0 pues ya tenemos el máximo de tokens). Con esta idea en mente, todas las votaciones comenzarán con el máximo de tokens permitidos en juego, y se tendrá que aprobar propuestas con estos, por lo que irán reduciendo su cantidad al ir ejecutándose y de esta forma estableciendo un límite de proposals que se pueden aprobar en una votación. Así aparece un motivo para cerrar una votación y abrir una nueva.

2.3. addParticipant

- En el enunciado se indica que es necesario comprar al menos un token para poder inscribirse. Pero como es posible que un participante transfiera tokens a alguien que no lo es, si el segundo quisiera inscribirse no consideramos tan necesario que compre, y por consiguiente, se obliga a tener algún token y no a comprarlo. Claro está que esta obligación implica la compra de tokens si no se disponía de ninguno previamente. Y si se llama a la función con un `msg.value` suficiente para comprar, también se compran tokens.
- En lo que resta, se calcula la cantidad de tokens a comprar con el `msg.value` (el sobrante se interpreta como donación a QV) y si QV dispone de esa cantidad de tokens. Por último se le transfieren los tokens al participante y se añade a las estructuras necesarias.

2.4. addProposal

- Un participante crea la proposal ccon los parámetros más el `msg.sender` como creador con la votación abierta.
- Se añade a las estructuras necesarias asignándole un id que es el valor a devolver.
- Además, registramos si es de *signaling* o *finance* como información adicional para `getSignaling` y `getPending`.

2.5. cancelProposal

- Solo el participante que creara la proposal puede cancelarla si no ha sido aprobada.
- A cada participante que la votara se le devuelven los votos y tokens asociados, y se borra la estructura.
- Además, se indica como cancelada y se restan tokens y votos hasta 0.

2.6. buyTokens

- Igual que en `addParticipant`, se calculan los tokens a comprar con el `msg.value` comprobando que al menos se compra uno y si QV dispone de esa cantidad de tokens.
- Luego, se le transfieren los tokens al participante.

2.7. sellTokens

- Se comprueba que el participante tiene algún token para luego transferirlos del participante a QV. Nótese que se necesita *allowance* previo, pero no aparece aquí el `require` porque está dentro de la función `spendAllowance` del ERC20 que es llamado por `transferFrom`.
- QV le transfiere al participante el precio equivalente.

2.8. getERC20Voting

- Devuelve la dirección del contrato ERC20 que gestiona los tokens, para que los participantes interactúen externamente con este.

2.9. getPendingProposals

- Recorre las proposals guardando los índices de las de tipo *finance*. La variable `pendingCount` guarda cuantas proposals de *finance* hay.

2.10. getSignalingProposals

- Recorre las proposals guardando los índices de las de tipo *signaling* que detecta por tener budget 0. La variable `signalingCount` guarda cuantas proposals de *signaling* hay.

2.11. getApprovedProposals

- Devuelve una copia en `memory` del array `proposalsApproved` que va guardando los índices de las propuestas aprobadas.

2.12. getProposalInfo

- Devuelve una proposal dado su id (en `memory`). Se comprueba que el id tiene, en verdad, una proposal asociada.

2.13. stake

- Acción de votar.
- Se necesita ser participante, que la votación esté abierta, que sea una proposal válida todavía no aprobada ni cancelada.
- Se necesita dar al menos un voto.
- Se calcula cuantos tokens cuestan los votos añadidos atendiendo a la fórmula cuadrática.
- Se comprueba que el participante dispone de esos tokens y si es así, se transfieren del participante a QV, necesitando de un *allowance* previo.
- En `participantVotes` se actualiza el número de votos del participante a la proposal en concreto, y en proposals se añaden también los votos junto a los tokens que conlleva.
- Si la proposal es de tipo *finance* se procede a comprobar si está en condiciones de poder ejecutarse.

2.14. withdrawFromProposal

- Acción de retirar un voto.
- Se necesita ser participante, que la votación esté abierta, que sea una proposal válida todavía no aprobada ni cancelada, que haya recibido algún voto de parte del participante.
- Debe retirar como mínimo un voto y como máximo los votos previos.
- Se le transfieren al participante los tokens asociados a los votos retirados teniendo en cuenta el coste cuadrático.
- En `participantVotes` se reducen los votos del participante a la proposal, y en proposals disminuyen tanto votos como tokens.

2.15. _checkAndExecuteProposal

- Se comprueba que se cumplen las condiciones para ejecutar una propuesta que no haya sido aprobada ni cancelada todavía.
- En primer lugar, que el presupuesto total sumado al valor que equivale a los tokens destinados a la proposal es suficiente para ejecutar la proposal, es decir, mayor o igual que el budget de la proposal.

- Después se comprueba que los votos superan el umbral

$$umbral = \left(0.2 + \frac{budget}{totalBudget}\right) \cdot numParticipants + numProposals$$

Como trabajamos con enteros modificamos para quitar las fracciones y el 0.2, teniendo en cuenta que tal multiplicación afecta al total de la desigualdad, en otras palabras, a los votos que se comparan con el umbral, que multiplicamos también por $10 \cdot totalBudget$.

$$umbral = (2 \cdot totalBudget + 10 \cdot budget) \cdot numParticipants + numProposals \cdot 10 \cdot totalBudget$$

- Ponemos la propuesta como aprobada, para evitar ataques por parte del contrato que ejecuta la propuesta.
- Si se cumplen las condiciones se ejecuta la propuesta llamando a la función `executeProposal` de la interfaz `IExecutableProposal`, el presupuesto total se actualiza, incrementa su valor según el importe recaudado por votos y lo decrementa por el budget que envía al contrato de la Proposal.
- Además, se indica que la propuesta ha sido aprobada tanto en el atributo del mapping como añadiéndola al array de propuestas aprobadas.
- Por último, se queman con `burn` los tokens dedicados a esta propuesta.
- Observación: utilizamos un `lock` por seguridad, para prevenir un posible ataque de *reentrancy* por parte de la función `executeProposal`.

2.16. closeVoting

- Se cierra la votación poniendo `votingOpen` a false.
- Para cada propuesta pendiente, que son las de *signaling* y las de *finance* no aprobadas ni canceladas, se devuelven los tokens a los participantes. En el caso de las signaling, además, se ejecuta la proposal con la función `executeProposal` del contrato asociado. Al `owner` se le transfiere el presupuesto restante (`totalBudget`).
- En cuanto a los mapping y otras variables, se borran las estructuras que registran los votos de los participantes y las proposals, y se restablecen los contadores a sus valores iniciales.

3. Prueba de ejecución

1. Deploy QuadraticVoting con precio = 2 y supply = 1000 con la cuenta A
2. Añadir participante con la cuenta B con value = 20 para comprar 10 tokens
3. Con la cuenta B hacer getERC20Voting y copiar el address para hacer deploy at address copiada
4. Aquí se pueden hacer comprobaciones con la cuenta B de las funciones view de ERC20
5. Añadir participante con la cuenta C con value = 40 para comprar 20 tokens
6. Hacemos openVoting con la cuenta A, con un value = 50, que se guarda en totalBudget
    - La cuenta A es el owner del contrato
    - El balance de QV es 110 (50 de totalBudget + 60 de 30 tokens comprados)
7. Hacemos un deploy de ProposalContract
8. Copiamos el address del contrato ProposalContract para añadir proposal con la cuenta B en Quadratic Voting. Elegimos un título (“a”) y descripción (“b”), y un budget de 60 (coste de ejecutar la propuesta).
    - Observación: si se hubiera hecho antes del openVoting (paso 6) haría revert por no estar abierta la votación.
9. Ahora se podrían hacer comprobaciones de get de las propuestas para ver que solo hay una pendiente, o para ver la información de la propuesta 1 (que es la que se acaba de añadir).
10. Hacemos otro deploy de ProposalContract
11. Con la cuenta C, añadirmos otra propuesta de Signaling (título = “s”, descripción “t”, amount = 0, address = la obtenida en el paso 10)
12. Ahora se podrían hacer comprobaciones de get de las propuestas para ver que ahora también hay una signaling.
13. Para poder votar, hay que hacer un approve en el contrato ERC20 de la cuenta B a QuadraticVoting con amount el número de votos para permitir llevar a cabo esos votos. En nuestro caso lo hacemos con amount = 1.
14. Votamos con la cuenta B haciendo stake al propId = 2, y newVotes = 1.
15. Con la cuenta C hacemos approve de 9 a QuadraticVoting.

16. Con la cuenta C hacemos un stake con 3 votos a la propuesta 2, que equivalen a los 9 tokens permitidos en el paso anterior.

17. Podemos ver ahora la información de la propuesta 2 que tiene 4 votos y 10 tokens: 1 voto $\leftrightarrow$ 1 token y 3 votos $\leftrightarrow$ 9 tokens.

18. Se puede comprobar en ERC20 las fluctuaciones de los tokens con las funciones allowance y balanceOf.

19. Con la cuenta C, retiramos 1 voto de la propuesta 2.

20. Se puede comprobar la información de la propuesta 2 que pasa a 3 votos y 5 tokens: 1 voto $\leftrightarrow$ 1 token, y 2 votos $\leftrightarrow$ 4 tokens

21. Se puede comprobar en ERC20 las fluctuaciones de los tokens con las funciones allowance y balanceOf.

22. Con la cuenta B, damos 1 voto a la proposal 1. Recordar que hay que hacer un approve, por ejemplo de 2 tokens.

23. El estado actual de los balances en ERC20 serían:

    - QV: 976
    - B: 8 (que es los 10 iniciales - 1 voto a propuesta1 - 1 voto a propuesta2)
    - C: 16 (que es 20 iniciales - 2 votos a la misma propuesta)

24. Con la cuenta C, hacemos un cancelProposal de la propuesta 2.

25. Se puede comprobar en ERC20 con balanceOf que se devuelven a la cuenta B un token, y a la cuenta C 4 tokens.

26. Con la cuenta C volvemos a añadir la proposal de signalin que acababa de cancelar. Ahora tendría identificador 3.

27. Ahora queremos ejecutar la propuesta 1. Para ello tenemos que tener en cuenta el threshold:

    - $(0.2 + 60/50) \cdot 2 + 2 = 4.8$ (Hay que llegar a 5 o mas votos para ejecutarla)
    - $50 + 13 * 2 = 76 > 60$
    - Vamos a dar 2 votos con la cuenta C a la propuesta 1 (cuesta 4 tokens)
    - Con la cuenta B damos 2 votos (cuestan 8 tokens)
    - Así, se superarían las dos condiciones y se ejecuta la propuesta. El totalBudget se actualiza a 16.
    - El balance de QV sería 50 (110 que tenía - 60 de la propuesta). (Desde otro punto de vista son 16 del totalBudget + (1token*2precio) + (16tokens*2precio) que suman 50).

28. En la información de la propuesta 1, ha cambiado el valor de approve a true
29. getApprovedProp devuelve la propuesta 1 y getPendingProp deja de devolverla
30. Se pueden observar el evento de ejecutarse la propuesta en el log.

    “from”: “0x417Bf7C9dc415FEEb693B6FE313d1186C692600F”,

    “topic”: “0x040e4621d86c508ac4e21bb969b0de47290445bc4dd7aaeef5e780bb9ad6c1a1”,

    “event”: “Execute”,

    “args”: {

    “0”: “1”,

    “1”: “5”,

    “2”: “13”,

    “_proposalId”: “1”,

    “_numVotes”: “5”,

    “_numTokens”: “13”

    }
31. Por la función getBalance incorporada al contrato ProposalContract que implementa IExecutableProposal, se comprueba que devuelve un valor de 60, que era el budget necesario para ejecutarla.
32. El estado actual del ERC20 sería:

    - QV: 970
    - B: 1
    - C: 16

    Podemos observar que la suma es 987 (totalSupply), y restan hasta 1000 los 13 que se han empleado en la ejecución de la propuesta y que se han hecho burn.
33. Con la cuenta C damos 1 voto a la propuesta 3.
34. Con la cuenta B quiere dar dos votos, pero solo dispone de 1 token, así que hacemos que compre con buyTokens 5 tokens.
35. Con la cuenta B damos 2 votos a la propuesta 3.
36. Le sobrarían 2 tokens que quiere vender, así que para hacer sellTokens necesita hacer primero approve, al igual que se hacía en stake al votar.
37. Ahora vamos a cerrar la votación. Con la cuenta A hacemos closeVoting.
38. A la cuenta A, se le han añadido los 16 weis del totalBudget que quedaban al cerrar la votación.

39. Observamos en ERC20 con balanceOf que la cuenta B recupera 4 tokens, y la cuenta C recupera 1 token, quedándose al final con 16. QuadraticVoting se queda con 967, que sumados a los de los participantes, son 987 como tenía en paso 32*.

40. También podemos ver el execute de la propuesta de signaling que se ha ejecutado.

    “from”: “0x4815A8Ba613a3eB21A920739dE4cA7C439c7e1b1”,

    “topic”: “0x040e4621d86c508ac4e21bb969b0de47290445bc4dd7aaeef5e780bb9ad6c1a1”,

    “event”:“Execute”,

    “args”: {

    “0”: “3”,

    “1”: “3”,

    “2”: “5”,

    “_proposalId”: “3”,

    “_numVotes”: “3”,

    “_numTokens”: “5”

    }

41. Vamos a ver ahora que el contrato ERC20 lo pueden seguir ejecutando desde fuera de QV con la votación cerrada.

42. A una cuenta D, B le transfiere 3 tokens que tenía y la cuenta B venderá 1 restante.

43. Con balanceOf se comprueba que efectivamente se han transferido.

44. La cuenta D hace addParticipant sin comprar tokens, ya que dispone de 3.

45. Finalmente, con la cuenta A vuelve a abrir la votación, y si vemos en ERC20 el total supply han vuelto a tener 1000.

Anexos

A. ERC20

```
1
2  abstract contract Context {
3      function _msgSender() internal view virtual returns (address) {return
           msg.sender;}
4      function _msgData() internal view virtual returns (bytes calldata) {
           return msg.data;}
5  }
6  interface IERC20 {
7      event Transfer(address indexed from, address indexed to, uint256 value
           );
8      event Approval(address indexed owner, address indexed spender, uint256
            value);
9      function totalSupply() external view returns (uint256);
10     function balanceOf(address account) external view returns (uint256);
11     function transfer(address to, uint256 amount) external returns (bool);
12     function allowance(address owner, address spender) external view
           returns (uint256);
13     function approve(address spender, uint256 amount) external returns (
           bool);
14     function transferFrom(address from, address to, uint256 amount)
           external returns (bool);
15 }
16 interface IERC20Metadata is IERC20 {
17     function name() external view returns (string memory);
18     function symbol() external view returns (string memory);
19     function decimals() external view returns (uint8);
20 }
21 contract ERC20 is Context, IERC20, IERC20Metadata{
22     mapping(address => uint256) private _balances;
23     mapping(address => mapping(address => uint256)) private _allowances;
24     uint256 private _totalSupply;
25     string private _name;
26     string private _symbol;
27     constructor(string memory name_, string memory symbol_) {
28         _name = name_;
29         _symbol = symbol_;
30     }
31     function name() public view virtual override returns (string memory) {
           return _name;}
32     function symbol() public view virtual override returns (string memory)
           {return _symbol;}
33     function decimals() public view virtual override returns (uint8) {
           return 18;}
34     function totalSupply() public view virtual override returns (uint256)
           {return _totalSupply;}
```

```
35      function balanceOf(address account) public view virtual override
            returns (uint256) {return _balances[account];}
36
37      function transfer(address to, uint256 amount) public virtual override
            returns (bool) {
38          address owner = _msgSender();
39          _transfer(owner, to, amount);
40          return true;
41      }
42      function allowance(address owner, address spender) public view virtual
             override returns (uint256) {
43          return _allowances[owner][spender];
44      }
45      function approve(address spender, uint256 amount) public virtual
            override returns (bool) {
46          address owner = _msgSender();
47          _approve(owner, spender, amount);
48          return true;
49      }
50      function transferFrom(address from,address to,uint256 amount) public
            virtual override returns (bool) {
51          address spender = _msgSender();
52          _spendAllowance(from, spender, amount);
53          _transfer(from, to, amount);
54          return true;
55      }
56      function increaseAllowance(address spender, uint256 addedValue) public
             virtual returns (bool) {
57          address owner = _msgSender();
58          _approve(owner, spender, allowance(owner, spender) + addedValue);
59          return true;
60      }
61      function decreaseAllowance(address spender, uint256 subtractedValue)
            public virtual returns (bool) {
62          address owner = _msgSender();
63          uint256 currentAllowance = allowance(owner, spender);
64          require(currentAllowance >= subtractedValue, "ERC20: decreased
                allowance below zero");
65          unchecked {
66              _approve(owner, spender, currentAllowance - subtractedValue);
67          }
68          return true;
69      }
70      function _transfer(address from, address to, uint256 amount) internal
            virtual {
71          require(from != address(0), "ERC20: transfer from the zero address
                ");
72          require(to != address(0), "ERC20: transfer to the zero address");
73          _beforeTokenTransfer(from, to, amount);
74          uint256 fromBalance = _balances[from];
75          require(fromBalance >= amount, "ERC20: transfer amount exceeds
```

```
            balance");
        unchecked {
            _balances[from] = fromBalance - amount;
        }
        _balances[to] += amount;
        emit Transfer(from, to, amount);
        _afterTokenTransfer(from, to, amount);
    }
    function _mint(address account, uint256 amount) internal virtual {
        require(account != address(0), "ERC20: mint to the zero address");
        _beforeTokenTransfer(address(0), account, amount);
        _totalSupply += amount;
        _balances[account] += amount;
        emit Transfer(address(0), account, amount);
        _afterTokenTransfer(address(0), account, amount);
    }
    function _burn(address account, uint256 amount) internal virtual {
        require(account != address(0), "ERC20: burn from the zero address"
            );
        _beforeTokenTransfer(account, address(0), amount);
        uint256 accountBalance = _balances[account];
        require(accountBalance >= amount, "ERC20: burn amount exceeds
            balance");
        unchecked {
            _balances[account] = accountBalance - amount;
        }
        _totalSupply -= amount;
        emit Transfer(account, address(0), amount);
        _afterTokenTransfer(account, address(0), amount);
    }
    function _approve(address owner, address spender, uint256 amount)
        internal virtual {
        require(owner != address(0), "ERC20: approve from the zero address
            ");
        require(spender != address(0), "ERC20: approve to the zero address
            ");
        _allowances[owner][spender] = amount;
        emit Approval(owner, spender, amount);
    }
    function _spendAllowance(address owner, address spender, uint256
        amount) internal virtual {
        uint256 currentAllowance = allowance(owner, spender);
        if (currentAllowance != type(uint256).max) {
            require(currentAllowance >= amount, "ERC20: insufficient
                allowance");
            unchecked {
                _approve(owner, spender, currentAllowance - amount);
            }
        }
    }
    function _beforeTokenTransfer(address from, address to, uint256 amount
```

```
        ) internal virtual {}
    function _afterTokenTransfer(address from, address to, uint256 amount)
        internal virtual {}
}
contract ERC20TC is ERC20{
    uint256 private immutable _cap;
    address private _controller;
    constructor(string memory name_, string memory symbol_,uint256 cap_)
        ERC20(name_,symbol_) {
        require(cap_ > 0, "ERC20Capped: cap is 0");
        _cap = cap_;
        _controller = msg.sender;
    }
    function cap() public view virtual returns (uint256) {return _cap;}
    function controller() public view virtual returns (address) {return
        _controller;}
    function mint(uint256 amount) public virtual returns (bool) {
        require(msg.sender==_controller,"Only QuadraticVoting has
            permision to mint");
        require(ERC20.totalSupply() + amount <= _cap, "ERC20Capped: cap
            exceeded");
        _mint(msg.sender, amount);
        return true;
    }
    function burn(uint256 amount) public virtual returns (bool) {
        require(msg.sender==_controller,"Only QuadraticVoting has
            permision to burn");
        _burn(msg.sender, amount);
        return true;
    }
}
```

B. IExecutableProposal

```
interface IExecutableProposal {
    function executeProposal(uint proposalId, uint numVotes, uint
        numTokens) external payable;
}
contract ProposalContract is IExecutableProposal {
    event Execute(uint _proposalId, uint _numVotes,uint _numTokens);
    function executeProposal(uint proposalId, uint numVotes,uint numTokens
        ) override external payable {
        emit Execute(proposalId,numVotes,numTokens);
    }
    function getBalance() public view returns(uint){
        return address(this).balance;
    }
}
```

C. Quadratic Voting Contract

```
1  // SPDX-License-Identifier: GPL-3.0
2  pragma solidity ^0.8.0;
3
4  //Javier Mulero Martin y Jorge del Valle Vazquez
5
6  contract QuadraticVoting{
7      // Struct de una propuesta
8      struct Proposal{
9          string title; // titulo
10         string description; // descripcion
11         uint256 budget; // presupuesto necesario para llevar a cabo la
               propuesta (0 en signalin)
12         address contractProposal; // direccion de un contrato que
               implementa IExecutableProposal
13         address creator; // creador
14         bool approved; // true si aprobada
15         bool canceled; // true si cancelada
16         uint tokens; // numero de tokens en la propuesta
17         uint votes; // numero de votos en la propuesta
18     }
19
20     address payable owner; // Propietario de la votacion
21     ERC20TC tokenContract; // Contrato ERC20
22     bool private votingOpen; // True si votacion abierta
23     bool private lock; // Lock para seguridad
24     uint private idNext; // Identificador actual de propuestas (va
           aumentando)
25     uint private signalingCount; // Numero de propuestas signalin
26     uint private pendingCount; // Numero de propuestas pendientes de
           aprobar (de financiacion)
27     uint256 private tokenPrice; // Precio del token
28     uint256 private totalBudget; // ETH de tokens comprados
29     /* Obs: puede no coincidir con address(this).balance, pues los tokens
           de la propuestas signaling
30      * o que se cancelan son devueltos a sus propietarios al finalizar la
            votacion, donde el propietario
31      * recauda todo, pero el contrato tiene que mantenet el ETH de los
            tokens no gastados */
32
33     mapping (address => bool) private isParticipant; // Participante ->
           true si es participante
34     mapping (address => mapping (uint => uint)) private participantVotes;
            // Participante -> Proposal - Votos a la propuesta
35     address[] private participantIterator = new address[](0); // Para
           iterar participantVotes
36
37     mapping (uint => Proposal) private proposals; // id propuesta -> datos
            propuesta
38     uint[] private proposalsApproved = new uint[](0); // Indice de los
```

```
        proposals aprobados uint[]
39      uint[] private proposalsIterator = new uint[](0); // Para iterar
        proposals y su length == numProposals del threshold
40
41      /*
42      En la creacion del contrato se debe proporcionar el precio en Wei de
        cada token y el
43      numero maximo de tokens que se van a poner a la venta para votaciones.
         Entre otras
44      cosas, el constructor debe crear el contrato de tipo ERC20 que
        gestiona los tokens.
45      */
46
47      constructor(uint256 _price, uint256 _supply) {
48          owner = payable(msg.sender);
49          tokenPrice = _price;
50          tokenContract = new ERC20TC("Voting Token", "VT", _supply);
51          tokenContract.mint(_supply);
52          votingOpen = false;
53          idNext = 1;
54      }
55      modifier onlyOwner {
56          require(msg.sender == owner, "Solo para el propietario del
            contrato.");
57          _;
58      }
59      modifier isNewParticipant {
60          require(isParticipant[msg.sender]==false, "El participante ya
            incorporado");
61          _;
62      }
63      modifier enoughTokens {
64          require(tokenContract.balanceOf(address(this))>0, "Faltan tokens")
            ;
65          _;
66      }
67      modifier enoughPrice {
68           require(msg.value >=tokenPrice, "Faltan ethers");
69          _;
70      }
71      modifier onlyOpen {
72          require(votingOpen, "Votacion no iniciada");
73        _;
74      }
75      modifier onlyCreator(uint _id) {
76          require(msg.sender==proposals[_id].creator, "No eres el creador");
77        _;
78      }
79      modifier onlyParticipant {
80          require(isParticipant[msg.sender], "El participante no esta
            incorporado");
```

```
81          _;
82      }
83      modifier isProposal(uint _id) {
84          require(proposals[_id].creator!=address(0), "La propuesta no
                existe");
85          _;
86      }
87      modifier onlyPending(uint _id) {
88          require(proposals[_id].creator!=address(0) && !proposals[_id].
                approved && !proposals[_id].canceled, "propuesta aprobada o
                cancelada");
89          _;
90      }
91
92      function getBalance() public view returns(uint){
93          return address(this).balance;
94      }
95
96      /*
97      Apertura del periodo de votacion. Solo lo puede ejecutar el usuario
            que
98      ha creado el contrato. En la transaccion que ejecuta esta funcion se
            debe transferir el
99      presupuesto inicial del que se va a disponer para financiar propuestas
            . Recuerda que
100     este presupuesto total se modificara cuando se aprueben propuestas: se
             incrementara
101     con las aportaciones en tokens de los votos de las propuestas que se
            vayan aprobando
102     y se decrementara por el importe que se transfiere a las propuestas
            que se aprueben
103     */
104
105     function openVoting() public payable onlyOwner{
106         require(!votingOpen, "openVoting: La votacion ya ha sido iniciada"
                );
107         require(msg.value>0, "openVoting: Se necesita aportar un minimo de
                 presupuesto");
108         votingOpen = true;
109         totalBudget = msg.value;
110         // Reiniciamos el totalSupply al cap
111         tokenContract.mint(tokenContract.cap() - tokenContract.totalSupply
                ());
112     }
113
114     /*
115     Funcion que utilizan los participantes para inscribirse en la votacion
            .
116     Los participantes se pueden inscribir en cualquier momento, incluso
            antes de que se abra el periodo de votacion.
117     Cuando se inscriben, los participantes deben transferir Ether para
```

```
        comprar tokens (al menos un token) que utilizaran para realizar sus votaciones.
118     Esta funcion debe crear y asignar los tokens que se pueden comprar con ese importe.
119     */
120
121     function addParticipant() external payable isNewParticipant {
122         // Deben tener dinero para comprar un token, o si ya eran participantes de una votacion
123         // de este mismo contrato, tienen que tener tokens
124         require(msg.value >= tokenPrice || tokenContract.balanceOf(msg.sender) > 0, "addParticipant: Todo participante necesita de tokens");
125         // Cantidad de tokens compra
126         uint256 tokenBought = msg.value / tokenPrice; // Aqui se queda la propina (msg.value % tokenPrice)
127
128         // No puede superar la cantidad disponible de tokens
129         require(tokenContract.balanceOf(address(this))>=tokenBought, "addParticipant: No hay mas tokens disponibles");
130
131         isParticipant[msg.sender] = true;
132         // Transferir los tokens
133         tokenContract.transfer(msg.sender, tokenBought);
134         participantIterator.push(msg.sender);
135         totalBudget += msg.value % tokenPrice; // El exceso o cambio se dedica a apoyar las proposals
136     }
137
138     /*
139     Funcion que crea una propuesta. Cualquier participante puede crear propuestas, pero solo cuando la votacion esta abierta.
140     Recibe todos los atributos de la propuesta: titulo, descripcion, presupuesto necesario para llevar a cabo la propuesta
141     (puede ser cero si es una propuesta de signaling) y la direccion de un contrato que implemente el interfaz ExecutableProposal,
142     que seria el receptor del dinero presupuestado en caso de ser aprobada la propuesta.
143     Debe devolver un identificador de la propuesta creada.
144     */
145
146     function addProposal(string memory _title, string memory _description, uint256 _amount, address _contractProposal) onlyParticipant onlyOpen external returns (uint256){
147         uint propId = (idNext++);
148         proposals[propId] = Proposal(_title,_description,_amount,_contractProposal,msg.sender,false,false,0,0);
149         proposalsIterator.push(propId);
150         if(_amount == 0)
151             signalingCount++;
152         else
```

```
153             pendingCount++;
154         return propId;
155     }
156 
157     /*
158     Cancela una propuesta dado su identificador. Solo se puede ejecutar si
            la votacion esta abierta. El unico que puede realizar esta accion
159     es el creador de la propuesta. No se pueden cancelar propuestas ya
            aprobadas. Los tokens recibidos hasta el momento para votar la
            propuesta
160     deben ser devueltos a sus propietarios.
161     */
162 
163     function cancelProposal(uint256 _propId) external onlyOpen onlyCreator
            (_propId) isProposal(_propId) onlyPending(_propId){
164         uint l = participantIterator.length;
165         address it;
166         uint votes;
167         // Devolver los tokens de los votantes de la propuesta (recorremos
                todos los participantes para ver quien ha votado)
168         for(uint i = 0; i < l; ++i){
169             it = participantIterator[i];
170             votes = participantVotes[it][_propId];
171             if(votes > 0){
172                 // El participante ha votado a la proposal => devolvemos
                        tokens
173                 tokenContract.transfer(it, votes**2);
174                 participantVotes[it][_propId] = 0; // Ya no tiene ningun
                        voto asociada
175                 delete participantVotes[it][_propId]; // Borramos el
                        mapping para el participante it
176             }
177         }
178 
179         proposals[_propId].canceled = true;
180         proposals[_propId].tokens = 0;
181         proposals[_propId].votes = 0;
182         if(proposals[_propId].budget == 0) signalingCount--;
183         else pendingCount--;
184     }
185 
186     /*
187     Esta funcion permite a un participante ya inscrito comprar mas tokens
            para depositar votos.
188     */
189 
190     function buyTokens()  onlyParticipant enoughTokens enoughPrice
            external payable {
191         uint256 tokenBought = msg.value / tokenPrice;
192         require(tokenContract.balanceOf(address(this))>=tokenBought, "
                buyTokens: No hay mas tokens disponibles");
```

```
193         tokenContract.transfer(msg.sender, tokenBought);
194     }
195
196     /*
197     Operacion complementaria a la anterior: permite a un participante
        devolver tokens no gastados en votaciones y recuperar el dinero
        invertido en ellos.
198     */
199
200     function sellTokens() onlyParticipant external {
201         uint tokenSell =  tokenContract.balanceOf(msg.sender); // Tokens a
            vender
202         require(tokenSell > 0, "sellTokens: No dispones de tokens a vender
            ");
203         tokenContract.transferFrom(msg.sender, address(this), tokenSell);
            //***necesita allowance? el mismo que en stake
204         uint256 price = tokenSell*tokenPrice;
205         require(address(this).balance >= price, "SellTokens: El contrato
            QuadraticVoting no dispone de suficente balance para devoler");
206         payable(msg.sender).transfer(price);
207     }
208
209     /*
210     Devuelve la direccion del contrato ERC20 que utiliza el sistema de
        votacion para gestionar tokens.
211     De esta forma, los participantes pueden utilizarlo para operar con los
         tokens comprados (transferirlos, cederlos, etc.).
212     */
213
214     function getERC20Voting()  public view onlyParticipant returns (
        address) {
215         return address(tokenContract);
216     }
217
218     /*
219     Devuelve un array con los identificadores de todas las propuestas de
        signaling (las que se han creado con presupuesto cero).
220     Solo se puede ejecutar si la votacion esta abierta.
221     */
222
223     function getSignalingProposals() public view onlyOpen returns (uint256
        [] memory){
224         uint[] memory sigPro = new uint[](signalingCount);
225         uint propId;
226         uint l = proposalsIterator.length;
227         uint j = 0;
228         for(uint i = 0; i < l ; i++){
229             propId = proposalsIterator[i];
230             //***Si se ejecutan solo en close voting sobra el approve
231             if (proposals[propId].budget == 0 && !proposals[propId].
                canceled && !proposals[propId].approved) {
```

```
232                 sigPro[j] = propId;
233                 j++;
234             }
235         }
236         return sigPro;
237     }
238 
239     /*
240     Devuelve un array con los identificadores de todas las propuestas
        pendientes de aprobar. Solo se puede ejecutar si la votacion esta
        abierta.
241     */
242 
243     function getPendingProposals() public view onlyOpen returns (uint256[]
        memory) {
244         uint256[] memory pending = new uint[](pendingCount);
245         uint propId;
246         uint l = proposalsIterator.length;
247         uint j = 0;
248         // Recorremos las propuestas y vemos cuales son pendientes
249         for(uint256 i = 0; i < l ; i++){
250             propId = proposalsIterator[i];
251             // Las proposals pendientes son aquellas de financiacion que
                no han sido canceladas ni aprobadas
252             if(proposals[propId].budget!=0 && !proposals[propId].canceled
                && !proposals[propId].approved) {
253                 pending[j] = propId;
254                 j++;
255             }
256         }
257         return pending;
258     }
259 
260     /*
261     Devuelve un array con los identificadores de todas las propuestas
        aprobadas. Solo se puede ejecutar si la votacion esta abierta.
262     */
263 
264     function getApprovedProposals() public view onlyOpen returns (uint256
        [] memory) {
265         uint l = proposalsApproved.length;
266         // Copia del array
267         uint[] memory approved = new uint[](l);
268         for(uint i = 0; i < l ; i++){
269             approved[i] = proposalsApproved[i];
270         }
271         return approved;
272     }
273 
274     /*
275     Devuelve los datos asociados a una propuesta dado su identificador.
```

```
        Solo se puede ejecutar si la votacion esta abierta.
276     */
277
278     function getProposalInfo(uint256 _propId) public view onlyOpen
            isProposal(_propId) returns (Proposal memory) {
279         Proposal memory pro = proposals[_propId];
280         return pro;
281     }
282
283     /*
284     Recibe un identificador de propuesta y la cantidad de votos que se
            quieren depositar y realiza el voto del participante que invoca
            esta funcion.
285     Calcula los tokens necesarios para depositar los votos que se van a
            depositar, comprueba que el participante posee los suficientes
            tokens
286     para comprar los votos y que ha cedido (con approve) el uso de esos
            tokens a la cuenta del contrato de la votacion.
287     Recuerda que un participante puede votar varias veces (y en distintas
            llamadas a stake) una misma propuesta con coste total cuadratico.
288
289     El codigo de esta funcion debe transferir la cantidad de tokens
            correspondiente desde la cuenta del participante a la cuenta de
            este contrato
290     para poder operar con ellos. Como esta transferencia la realiza este
            contrato, el votante debe haber cedido previamente con approve los
            tokens
291     correspondientes a este contrato (esa cesion no se debe programar en
            QuadraticVoting: la debe realizar el participante con el contrato
            ERC20, que puede obtener con getERC20).
292     */
293
294     function stake(uint256 _propId, uint256 newVotes) public onlyOpen
            onlyParticipant isProposal(_propId) onlyPending(_propId){
295         require(newVotes >= 1, "stake: Se necesita al menos un voto para
                votar");
296         uint prevVotes = participantVotes[msg.sender][_propId];// Los
                votos previos del participante a la propuesta
297         // (nuevos+antiguos)**2-(antiguos)**2 = coste anadido por nuevos
298         uint tokenCost = (newVotes+prevVotes)**2 - prevVotes**2;
299         // El participante debe poseer los suficientes tokens para comprar
                los votos
300         require(tokenContract.balanceOf(msg.sender)>= tokenCost, "stake:
                No tienes tokens disponibles");
301         // ha cedido (con approve) el uso de esos tokens a la cuenta del
                contrato de la votacion
302         // ***esta comprobacion se hace ya en ERC20
303         // require(tokenContract.allowance(msg.sender, address(this)) >=
                tokenCost, "Excede los tokens permitidos");
304         tokenContract.transferFrom(msg.sender, address(this) , tokenCost);
305         participantVotes[msg.sender][_propId] = participantVotes[msg.
```

```
            sender][_propId] + newVotes;
306         proposals[_propId].tokens += tokenCost;
307         proposals[_propId].votes += newVotes;
308         // Solo se debe recalcular el umbral de una propuesta cada vez que
                reciba votos.
309         // Las signaling no se ejecutan aqui, solo se votan
310         if(proposals[_propId].budget != 0 ){
311             if(_checkAndExecuteProposal(_propId)){
312                 pendingCount--;
313             }
314         }
315     }
316
317     /*
318     Dada una cantidad de votos y el identificador de la propuesta, retira
           (si es posible) esa cantidad de votos depositados por el
           participante
319     que invoca esta funcion de la propuesta recibida. Un participante solo
            puede retirar de una propuesta votos que el haya depositado
           anteriormente
320     y si la propuesta no ha sido aprobada todavia. Recuerda que debes
           devolver al participante los tokens que utilizo para depositar
321     los votos que ahora retira (por ejemplo, si habia depositado 4 votos a
            una propuesta y retira 2, se le deben devolver 12 tokens).
322     */
323
324     function withdrawFromProposal(uint256 _propId, uint256 retireVotes)
           public onlyOpen onlyParticipant isProposal(_propId) onlyPending(
           _propId){
325         uint prevVotes = participantVotes[msg.sender][_propId]; // los
               votos previos del participante a la propuesta
326         require(prevVotes>0,"withdrawFromProposal: No hay votos previos");
327         require((retireVotes >= 1) && (retireVotes <= prevVotes), "
               withdrawFromProposal: Se debe retirar como minimo un voto y
               como mixiomo los votos previos");
328         uint256 tokenCost = prevVotes**2 - (prevVotes - retireVotes)**2;
329         tokenContract.transfer(msg.sender, tokenCost);
330         participantVotes[msg.sender][_propId] = participantVotes[msg.
               sender][_propId] - retireVotes;
331         proposals[_propId].tokens -= tokenCost;
332         proposals[_propId].votes -= retireVotes;
333     }
334
335     /*
336     Funcion interna que comprueba si se cumplen las condiciones para
           ejecutar la propuesta y la ejecuta utilizando la funcion
           executeProposal del contrato
337     externo proporcionado al crear la propuesta. En esta llamada debe
           transferirse a dicho contrato el dinero presupuestado para su
           ejecucion.
338     Recuerda que debe actualizarse el presupuesto disponible para
```

```
        propuestas (y no olvides anadir al presupuesto el importe recibido
            de los tokens de votos
339     de la propuesta que se acaba de aprobar). Ademas deben eliminarse los
            tokens asociados a los votos recibidos por la propuesta, pues la
            ejecucion de la propuesta los consume.
340
341     Cuando se realice la llamada a executeProposal del contrato externo,
            se debe limitar la cantidad maxima de gas que puede utilizar
342     para evitar que la propuesta pueda consumir todo el gas de la
            transaccion. Esta llamada debe consumir como maximo 100000 gas.
343     */
344     /*
345
346     Una propuesta i es aprobada si se cumplen dos condiciones:
347     (1) el presupuesto del contrato de votacion mas el importe recaudado
            por los votos recibidos es suficiente para financiar la propuesta
348     (2) el numero de votos recibidos supera un umbral
349     */
350
351     // Solo se emplea para propuestas de financiacion
352
353     function _checkAndExecuteProposal(uint _propId) isProposal(_propId)
            onlyPending(_propId) internal returns (bool){
354         require(!lock, "LOCK: Bloqueado, ya hay una ejecucion en curso");
355         // Para trabajar con enteros cambiamos la formula
356         // Multiplicamos todo por 10*totalBudget y asi trabajar con
                enteros
357         uint threshold = (2*totalBudget + 10*proposals[_propId].budget) *
                (participantIterator.length) + ((pendingCount+signalingCount)
                *10*totalBudget);
358
359         if(totalBudget + proposals[_propId].tokens*tokenPrice >= proposals
                [_propId].budget && proposals[_propId].votes*10*totalBudget >=
                threshold){
360             lock = true;
361
362             // Aprobamos la propuesta antes de ejecutarla
363             // para que executeProposal no intente atacar
364             proposals[_propId].approved = true;
365             IExecutableProposal(payable(proposals[_propId].
                    contractProposal)).executeProposal{value : proposals[
                    _propId].budget, gas :100000}(_propId, proposals[_propId].
                    votes,proposals[_propId].tokens);
366             // Al aprobarse se queda el valor de los tokens en el contrato
367             // el importe recaudado por los votos
368             totalBudget += proposals[_propId].tokens*tokenPrice;
369             // El presupuesto necesario para la proposal se resta del
                    disponible
370             totalBudget -= proposals[_propId].budget;
371             // proposals[_propId].approved = true; Lo ponemos antes
372             proposalsApproved.push(_propId);
```

```
373             //***burn
374             // solo puede hacer burn QuadraticVoting
375             tokenContract.burn(proposals[_propId].tokens);
376 
377             lock = false;
378             return true;
379         }
380         return false;
381     }
382 
383     /*
384     Cierre del periodo de votacion. Solo puede ejecutar esta funcion el
        usuario que ha creado el contrato de votacion. Cuando termina el
        periodo de votacion se deben
385     realizar entre otras las siguientes tareas:
386         - Las propuestas que no han podido ser aprobadas son descartadas y
            los tokens recibidos por esas propuestas es devuelto a sus
           propietarios.
387         - las propuestas de signaling son ejecutadas y los tokens
            recibidos mediante votos es devuelto a sus propietarios.
388         - El presupuesto de la votacion no gastado en las propuestas se
            transfiere al propietario del contrato de votacion.
389     Cuando se cierra el proceso de votacion no se deben aceptar nuevas
        propuestas ni votos y el contrato QuadraticVoting debe quedarse en
        un estado que permita abrir un nuevo proceso de votacion.
390     Esta funcion puede consumir una gran cantidad de gas, tenlo en cuenta
        al programarla y durante las pruebas.
391     */
392 
393     function closeVoting() public onlyOpen onlyOwner{
394         votingOpen = false; // Cerramos votacion
395         uint propId;
396         address partId;
397         //Las propuestas que no han podido ser aprobadas son descartadas y
            los tokens recibidos por esas propuestas es devuelto a sus
           propietarios
398         uint l=proposalsIterator.length;
399         uint le=participantIterator.length;
400         for(uint i = 0; i < l; i++){
401             propId = proposalsIterator[i];
402             if(!proposals[propId].canceled && !proposals[propId].approved)
                {
403                 for(uint j=0; j < le; j++){
404                     partId = participantIterator[j];
405                     uint votes = participantVotes[partId][propId];
406                     if(votes >0) {
407                         tokenContract.transfer(partId,votes**2);
408                     }
409                     //Las propuestas de signaling son ejecutadas y los
                        tokens recibidos mediante votoses devuelto a sus
                        propietarios(ya hecho por el if anterior)
```

```
                    if(proposals[propId].budget == 0) {
                        proposals[propId].approved = true;
                        IExecutableProposal(payable(proposals[propId].
                            contractProposal)).executeProposal{gas
                            :100000}(propId, proposals[propId].votes,
                            proposals[propId].tokens);
                    }
                }
            }
        }
        // El presupuesto de la votacion no gastado en las propuestas se
            transfiere al propietario del contrato de votacion.
        // EL balance del contrao es la suma / calculos de la cantidad
            aportada inicialmente mas los tokens en posesion de
            participantes
        owner.transfer(totalBudget);
        totalBudget = 0;
        // Borrado de estructuras (para dejar preparado QuadraticVoting
            para una nueva votacion)
        for(uint i = 0; i < le; i++){
            partId = participantIterator[i];
            for(uint j = 0; j < l; j++){
                propId = proposalsIterator[j];
                delete participantVotes[partId][propId];
            }
        }
        for(uint j=0; j < l; j++){
                propId = proposalsIterator[j];
                delete proposals[propId];
        }
        participantIterator = new address[](0);
        proposalsApproved = new uint[](0);
        proposalsIterator = new uint[](0);
        idNext = 1;
        signalingCount = 0;
        pendingCount = 0;
    }
}
```