PROYECTO FINAL

Alejandro Ramírez y David Seijas

May 2022

1. Código

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
pragma experimental ABIEncoderV2;

import "./SafeMath.sol";
import "./ERC20.sol";

interface IExecutableProposal {

    event Executing(uint id, uint numVotes, uint numTokens, uint budget, uint balance,
        string message);

    function executeProposal(uint proposalId, uint numVotes, uint numTokens) external
        payable;
}


contract SignalingProposal is IExecutableProposal {

    function executeProposal(uint proposalId, uint numVotes, uint numTokens) external
        payable override{
        emit Executing(proposalId, numVotes, numTokens, msg.value, address(this).balance, "
            Ejecutando la propuesta...");
    }
}

contract FinantialProposal is IExecutableProposal {

    function executeProposal(uint proposalId, uint numVotes, uint numTokens) external
        payable override{
        emit Executing(proposalId, numVotes, numTokens, msg.value, address(this).balance, "
            Ejecutando la propuesta...");
    }
}


contract TokenManager is ERC20 {

    address immutable admin;

    modifier onlyAdmin(){
        require(msg.sender == admin , "No permission");
        _;
    }

    constructor(string memory _name, string memory _desc) ERC20(_name,_desc){
        admin = msg.sender;
    }

    function mint(address account, uint256 amount) external onlyAdmin {
        _mint(account, amount);
    }

```

```
48      function burn(address account, uint256 amount) external onlyAdmin{
49          _burn(account, amount);
50      }
51
52  }
53
54
55  contract QuadraticVoting {
56
57      //Creador de la votacion
58      address immutable admin;
59      //Precio del token
60      uint tokenPrice;
61      // N   m ximo de tokens que se ponen en venta
62      uint maxTokens;
63      //Contrato ERC-20
64      TokenManager tokenContract;
65      //Presupuesto de la votacion disponible para propuestas
66      uint totalBudget;
67      //Comprobar si el periodo de votaci n est  abierto
68      bool isOpen;
69
70      //Informacion sobre una propuesta
71      struct ProposalInfo{
72          address creator;
73          string tittle;
74          string description;
75          uint budget; //Presupuesto de la propuesta
76          uint votes;
77          uint tokens;
78          uint indexList; //Indice de la correspondiente lista de propuestas para eliminarla
                cuando se cancela
79          address proposalContract; //cuenta del contrato de una propuesta que se crea
80          bool isApproved; //Redundante, para no tener que recorrer todo el array de
                notApproved en los modifiers
81      }
82
83      //Ver que participantes estan inscritos
84      mapping(address => bool) participants;
85
86      //Lista de participantes inscritos
87      address[] participantsList;
88
89      //Asociar id con la info de propuesta
90      mapping(uint => ProposalInfo) proposalsInfo;
91
92      //Lista de propuestas (guardamos los ids)
93      uint[] signProps;
94      uint[] finPropsNotApproved;
95      uint[] finPropsApproved;
96
97      //Asociar a cada propuesta la cantidad de votos que lleva cada persona en esa propuesta
98      mapping(address => mapping(uint => uint)) participantVotes;
99
100     //Cambio que le sobra a cada participante al comprar tokens.
101     mapping(address => uint) participantExchange;
102
103     //Numero hist rico de propuestas, utilizado para dar id a cada una
104     uint idProposals;
105
106     //Evitar vulnerabilidad Reentrancy
107     bool lock = false;
108
109     //Ejecutar acciones solo si eres el administrador de la votacion
110     modifier onlyAdmin(){
111         require(admin == msg.sender, "You're not the admin of the votation.");
112         _;
113     }
114
115     //Ejecutar acciones solo si la votacion esta abierta
```

```
116     modifier isPeriodOpen(){
117         require(isOpen, "The votation isn't open. You can't do this action.");
118         _;
119     }
120 
121     //Ejecutar acciones solo si eres participante
122     modifier onlyParticipants(){
123         require(participants[msg.sender], "You're not a participant yet.");
124         _;
125     }
126 
127     //Ejecutar acciones sobre una propuesta solo si eres el creador de ella
128     modifier onlyCreator(uint proposalId){
129         require(proposalsInfo[proposalId].creator == msg.sender, "You're not the creator of
                this proposal.");
130         _;
131     }
132 
133     //Comprobar que el id es valido
134     modifier isProposal(uint _proposalId){
135         require(proposalsInfo[_proposalId].creator != address(0), "This proposal doesn't
                exist.");
136         _;
137     }
138 
139     //Comprobar que la propuesta todavia no ha sido aprobada
140     modifier isnotProposalApproved(uint _proposalId){
141         require(!proposalsInfo[_proposalId].isApproved, "This proposal has already been
                approved.");
142         _;
143     }
144 
145     //Evento de aprobar una propuesta
146     event ApprovedProposal(uint _proposalId, string _message);
147     //Evento de compra de tokens
148     event BoughtTokens(uint _tokens, string _message);
149 
150     //No obligamos pero la idea es lanzar el contrato con tokenPrice > 0
151     constructor(uint _tokenPrice, uint _maxTokens){
152         admin = msg.sender;
153         tokenPrice = _tokenPrice;
154         maxTokens = _maxTokens;
155         tokenContract = new TokenManager("Toks", "desc");
156         idProposals = 1;
157     }
158 
159     //Abrir un periodo de votacion
160     function openVoting() public onlyAdmin payable {
161         //Obligamos a que el budget de la votacion sea > 0 porque si no nunca se pueden
                aprobar propuestas
162         require(msg.value > 0, "You need to introduce budget");
163         //Establecemos el presupuesto inicial del contrato es la cantidad con la que abrimos
                la votacion
164         totalBudget = msg.value;
165         //Abrimos votacion
166         isOpen = true;
167     }
168 
169     //A adir nuevo participante
170     function addParticipant() public payable {
171         require(!participants[msg.sender], "You are already a participant");
172         //Obligamos a que adquieran al menos 1 token
173         uint tokens = _calculateTokens(msg.value, msg.sender);
174         require(tokens >= 1, "Not enough ethers to buy at least one token");
175         //Comprobar que quedan suficientes tokens a la aventa
176         //TotalSuply siempre va a ser menor o igual que maxTokens
177         require(maxTokens - tokenContract.totalSupply() >= tokens, "There are not enought
                available tokens");
178 
179         //Inscribimos al participante
```

```
180         participants[msg.sender] = true;
181         participantsList.push(msg.sender);
182         //Transferir los tokens al participante
183         tokenContract.mint(msg.sender, tokens);
184 
185         //Emitir evento de tokens comprados
186         emit BoughtTokens(tokens, "You have bought {tokens} tokens"); //checkear como se
                escribe eso bien
187     }
188 
189     //A adir nueva propuesta
190     function addProposal(string memory _title, string memory _description, uint _budget,
            address _accountContract) public isPeriodOpen onlyParticipants returns(uint) {
191         require(msg.sender != address(0), "The creator must be an account");
192         //A adir propuesta a su correspondiente lista
193         uint index;
194         if(_budget == 0){//Es signaling
195             index = signProps.length;
196             signProps.push(idProposals);
197         }
198         else { //Financial no approved
199             index = finPropsNotApproved.length;
200             finPropsNotApproved.push(idProposals);
201         }
202 
203         //Guardar la informacion de la propuesta
204         ProposalInfo memory proposal = ProposalInfo(msg.sender, _title, _description,
                _budget, 0, 0, index, _accountContract, false);
205         proposalsInfo[idProposals] = proposal;
206         unchecked{ //2^256 es mayor que el n   de   tomos   del universo. No creemos que hayan
                 tantas propuestas
207             idProposals += 1;
208         }
209 
210         return idProposals - 1;
211     }
212 
213     //Cancelar propuesta
214     function cancelProposal(uint _proposalId) public isPeriodOpen isProposal(_proposalId)
            isnotProposalApproved(_proposalId) onlyCreator(_proposalId) {
215         //Devolucion de tokens
216         for(uint i = 0; i < participantsList.length; ++i){
217             address participant = participantsList[i];
218             //Tokens que se le deben devolver (si no ha votado no se le devuelve nada)
219             uint tokens = participantVotes[participant][_proposalId]**2;
220             if(tokens > 0)
221                 tokenContract.transfer(participant, tokens);
222         }
223 
224         //Descartar la propuesta
225         if(proposalsInfo[_proposalId].budget == 0){ //La propuesta es signaling
226             //Guardar indices involucrados para evitar mas accesos
227             uint lastIndex;
228             unchecked{ //Sabemos que al ser signaling tiene que haber alguna propuesta en el
                     array sigProps (y no se ha cancelado anteriormente por el modifier
                    isProposal)
229                 lastIndex = signProps.length - 1;
230             }
231             uint deleteIndex = proposalsInfo[_proposalId].indexList;
232             //Cambiar   ndice   del   ltimo
233             proposalsInfo[signProps[lastIndex]].indexList = deleteIndex;
234             //Intercambiar elementos en el array y eliminar el   ltimo
235             signProps[deleteIndex] = signProps[lastIndex];
236             signProps.pop();
237         }
238         else{//La propuesta es financial, not approved
239             //Guardar indices involucrados para evitar mas accesos
240             uint lastIndex;
241             unchecked{ //Idem
242                 lastIndex = finPropsNotApproved.length - 1;
```

```
            }
            uint deleteIndex = proposalsInfo[_proposalId].indexList;
            //Cambiar  ndice  del  ltimo
            proposalsInfo[finPropsNotApproved[lastIndex]].indexList = deleteIndex;
            //Intercambiar elementos en el array y eliminar el  ltimo
            finPropsNotApproved[deleteIndex] = finPropsNotApproved[lastIndex];
            finPropsNotApproved.pop();
        }

        //Eliminamos informacion del mapping (address se pone a 0 y deja de ser propuesta)
        delete proposalsInfo[_proposalId];
    }


    //Comprar tokens
    function buyTokens() public onlyParticipants payable {
        //Permitimos que el msg.value sea menor que un tokenPrice, por si con el exchange
            suma 1 token
        uint tokens = _calculateTokens(msg.value, msg.sender);
        //Comprobar que quedan suficientes tokens a la aventa
        require(maxTokens - tokenContract.totalSupply() >= tokens, "There are not enought
            available tokens");
        //Asignarle los tokens comprados al participante
        tokenContract.mint(msg.sender, tokens);
        //Emitir evento de tokens comprados
        emit BoughtTokens(tokens, "You have bought {tokens} tokens"); //checkear como se
            escribe eso bien
    }

    //Vender tokens
    function sellTokens(uint _numTokens) public onlyParticipants payable {
        //Comprobar que los tokens que quiere vender de verdad los tiene
        require(_numTokens >= tokenContract.balanceOf(msg.sender), "Imposible to sell more
            tokens than you have");
        //Calculamos los ethers que le tocan a devolver
        uint ethersDevolved = _numTokens * tokenPrice;
        //Quemar los tokens que se han vendido
        tokenContract.burn(msg.sender, _numTokens);
        //Le devolvemos los ethers al propietario de los tokens
        payable(msg.sender).transfer(ethersDevolved);
    }

    //Devuelve el total de tokens que quedan disponibles en el sistema
    function getRestTokens() public view returns(uint){
        return maxTokens - tokenContract.totalSupply();
    }

    //Devuelve cambio de un usuario
    function getExchange() public view returns(uint){
        return participantExchange[msg.sender];
    }

    //El participante recupera su cambio
    function regainExchange() public onlyParticipants payable{
        uint exchange = participantExchange[msg.sender];
        participantExchange[msg.sender] = 0;
        payable(msg.sender).transfer(exchange);
    }


    //Obtener la direccion del contrato ERC20
    function getERC20() public view returns(address){
        return address(tokenContract);
    }

    //Devolver array con los ids de las propuestas pendientes de aprobar (solo finantial
        pues signaling no se pueden aprobar)
    function getPendingProposals() public view isPeriodOpen returns(uint[] memory){
        return finPropsNotApproved;
    }
```

```
308
309     //Devolver array con los ids de las propuestas finantial ya aprobadas
310     function getApprovedProposals() public view isPeriodOpen returns(uint[] memory){
311         return finPropsApproved;
312     }
313
314     //Devolver array con los ids de las propuestas signaling
315     function getSignalingProposals() public view isPeriodOpen returns(uint[] memory){
316         return signProps;
317     }
318
319     //Devolver la info de una propuesta segun su id
320     function getProposalInfo(uint _proposalId) public view isPeriodOpen isProposal(
            _proposalId) returns(ProposalInfo memory){
321         return proposalsInfo[_proposalId];
322     }
323
324     //Votar en una propuesta
325     function stake(uint _proposalId, uint _votes) public isPeriodOpen isProposal(_proposalId
            ) isnotProposalApproved(_proposalId){
326         require(_votes > 0, "You have to deposit at least 1 vote.");
327
328         //Cantidad de votos que ha hecho esa persona a esa propuesta previamente
329         uint previousVotes = participantVotes[msg.sender][_proposalId];
330         //Cantidad de tokens necesarios a pagar para votar en funcion de votos anteriores
331         uint needTokens = (_votes + previousVotes)**2 - previousVotes**2;
332
333         //Comprobar: participante tiene suficientes tokens y permisos de nuestro contrato
                para operar con ellos
334         require(tokenContract.balanceOf(msg.sender) >= needTokens, "You don't have enough
                tokens to vote");
335         require(tokenContract.allowance(msg.sender, address(this)) >= needTokens, "You need
                to allow us to operate with your tokens");
336
337         //Actualizar la info de tokens y votos del participantes que ha votado
338         participantVotes[msg.sender][_proposalId] += _votes;
339         //Transferir tokens de la cuenta del participante a la del contrato
340         tokenContract.transferFrom(msg.sender, address(this), needTokens);
341
342         //Actualizar la info de tokens y votos de la propuesta que ha votado
343         proposalsInfo[_proposalId].votes += _votes;
344         proposalsInfo[_proposalId].tokens += needTokens;
345
346         //Llama a check si la propuesta no es signaling para ver si hay que ejecutarla
347         if(proposalsInfo[_proposalId].budget != 0)
348             _checkAndExecuteProposal(_proposalId);
349     }
350
351     //Retirar votos de una propuesta
352     function withdrawFromProposal(uint _proposalId, uint _votes) public isPeriodOpen
            isProposal(_proposalId) isnotProposalApproved(_proposalId){
353         //Comprobar que ha depositado esa cantidad de votos previamente (no se pueden
                retirar mas votos de los depositados)
354         uint previousVotes = participantVotes[msg.sender][_proposalId];
355         require(_votes <= previousVotes, "You haven't deposit that votes amount.");
356
357         //Cantidad de tokens que tenemos que devolver
358         uint returnTokens = previousVotes**2 - (previousVotes - _votes)**2;
359
360         //Actualizar los votos y tokens del participante
361         unchecked{ //Hemos comprobado que _votes es como mucho previousVotes
362             participantVotes[msg.sender][_proposalId] -= _votes;
363         }
364         //Devolver tokens a la cuenta del participante
365         tokenContract.transfer(msg.sender, returnTokens);
366
367         //Actualizar la info de votos y tokens de la propuesta
368         unchecked{ //La propuesta ha de tener al menos esos valores pues si se retiran los
                votos es porque voto y la info se actualiza al votar
369             proposalsInfo[_proposalId].votes -= _votes;
```

```
            proposalsInfo[_proposalId].tokens -= returnTokens;
        }
    }

    //Cerrar la votacion y dejarla en un estado que pueda volver a ser abierta
    function closeVoting() public isPeriodOpen onlyAdmin {
        //Cerramos periodo de votacion para evitar que se ejecuten otras funciones mientras se realiza el proceso
        isOpen = false;

        //Descartar propuestas no aprobadas
        for(uint i = 0; i < finPropsNotApproved.length; ++i){
            uint proposalId = finPropsNotApproved[i];

            //Devolver tokens a participantes que han votado una propuesta descartada
            for(uint j = 0; j < participantsList.length; ++j){
                tokenContract.transfer(msg.sender, participantVotes[participantsList[i]][proposalId]**2);
            }

            //Borrar info de la propuesta
            delete proposalsInfo[proposalId];
        }

        //Reiniciar lista de propuestas no aprobadas
        finPropsNotApproved = new uint[](0);

        //Aprobar Signaling proposals y descartarla posteriormente (borrado de info)
        for(uint i = 0; i < signProps.length; ++i){
            uint proposalId = signProps[i];

            //Devolver tokens a participantes que han votado a una signaling proposal
            for(uint j = 0; j < participantsList.length; ++j){
                tokenContract.transfer(msg.sender, participantVotes[participantsList[i]][proposalId]**2);
                //Al poner los votos a 0 nos aseguramos que nadie pueda volver a recibir tokens que no son suyos ya
                participantVotes[participantsList[i]][proposalId] = 0;
            }

            //Presupuesto de esa propuesta
            uint budget = proposalsInfo[proposalId].budget;
            //Votos de esa propuesta
            uint votes = proposalsInfo[proposalId].votes;
            //Tokens de esa propuesta
            uint tokens = proposalsInfo[proposalId].tokens;

            //Ejecutar signaling proposal
            //No protegemos con lock pues closeVoting solo lo podemos llamar nosotros
            IExecutableProposal(payable(proposalsInfo[proposalId].proposalContract)).executeProposal{value: budget, gas :100000}(proposalId, votes, tokens);

            //Borrar info de la propuesta
            delete proposalsInfo[proposalId];
        }

        //Reiniciar lista de propuestas no aprobadas
        signProps = new uint[](0);

        //Borrar info de las propuestas aprobadas y reiniciar su lista
        for(uint i = 0; i < finPropsApproved.length; ++i){
            delete proposalsInfo[finPropsApproved[i]];
        }
        finPropsApproved = new uint[](0);

        //Devolver presupuesto restante al propietario y reiniciamos para una nueva votacion, de forma segura para que nadie robe dinero
        uint transferBudget = totalBudget;
        totalBudget = 0;
        payable(admin).transfer(transferBudget);
```

```
434     }
435
436     //--------- FUNCIONES INTERNAS ---------
437
438     //Ejecuta una propuesta si se cumplen las conidicones necesarias para su aprobacion
439     function _checkAndExecuteProposal(uint _proposalId) internal {
440         require(!lock, "This action is being already executed. You have to wait.");
441         //Presupuesto de esa propuesta
442         uint budget = proposalsInfo[_proposalId].budget;
443         //Votos de esa propuesta
444         uint votes = proposalsInfo[_proposalId].votes;
445         //Tokens de esa propuesta
446         uint tokens = proposalsInfo[_proposalId].tokens;
447
448         //1   condicion: comprobar que tenemos suficiente presupuesto (el de la votacion mas
                el de los votos)
449         if (totalBudget + tokenPrice * tokens > budget){ //Obligamos a ser > para que
                totalBudget nunca sea 0 porque hace que la votacion sea inconsistente
450             //Calculamos el umbral de aprobacion
451             uint threshold = (2 * totalBudget + 10 * budget) * participantsList.length + 10
                    * totalBudget * (finPropsNotApproved.length + signProps.length);
452
453             //2   condicion: N   de votos recibidos por la propuesta supera el umbral
454             if(10 * totalBudget * votes >= threshold){
455                 //Ponemos propuesta aprobada para que no haya ataques de reentrancy cuando
                        votan
456                 proposalsInfo[_proposalId].isApproved = true;
457
458                 //Ejecutar la propuesta
459                 lock = true; //Creemos que se podr an quitar, pero es mejorar asegurar que
                        no hay vulnerabilidades
460                 IExecutableProposal(payable(proposalsInfo[_proposalId].proposalContract)).
                        executeProposal{value: budget, gas :100000}(_proposalId, votes, tokens);
461                 lock = false;
462
463                 //La propuesta ha sido aprobada y ejecutada
464                 uint lastIndex;
465                 unchecked{ //Si la propuesta ha sido aprobada es porque en stake el array de
                        porpuestas no aprobadas tenia al menos una porpuesta (la aporbada ahora
                        )
466                     lastIndex = finPropsNotApproved.length - 1;
467                 }
468                 uint deleteIndex = proposalsInfo[_proposalId].indexList;
469                 //Cambiar  ndice  de la ultima propuesta del array de no aprobadas
470                 proposalsInfo[finPropsNotApproved[lastIndex]].indexList = deleteIndex;
471
472                 //Eliminar propuesta de lista de no aprobadas
473                 finPropsNotApproved[deleteIndex] = finPropsNotApproved[lastIndex];
474                 finPropsNotApproved.pop();
475
476                 //A adimos la propuesta al array de aprobadas con su indice actualizada
477                 proposalsInfo[_proposalId].indexList = finPropsApproved.length;
478                 finPropsApproved.push(_proposalId);
479
480                 //Emitir evento de aprobacion de propuesta
481                 emit ApprovedProposal(_proposalId, "This proposal has been approved.");
482
483                 //Eliminar tokens asociados a los votos de la propuesta
484                 tokenContract.burn(address(this), tokens);
485
486                 //Actualizamos el presupuesto disponible para propuestas (quitamos el de la
                        propuesta aprobada y a adimos el importe de los tokens pagado para esta
                        )
487                 unchecked{ //Es segura por el if de arriba
488                     totalBudget += tokenPrice * tokens - budget;
489                 }
490
491                 //No hace falta actualizar el  n   de votos o tokens en ProposalInfo pues en
                        withdrawAll se requiere que la propuesta no haya sido aprobada, al
                        aprobarse esta propuesta ya no sirve para nada, solo para ver el array
```

```
                    de propuestas aprobadas
            }
        }
    }

    //Calcular el numero de tokens que puede comprar un participante en funci n de los
        ethers aportados
    function _calculateTokens(uint256 weis, address _participant) internal returns(uint){
        //Almacenar en memory para no acceder a storage numerosas veces
        uint _tokenPrice = tokenPrice;

        // N   de tokens y cambio que le quedar a seg n lo pagado (no sabemos si son
            seguras por si tokenPrice es 0)
        uint tokens = weis / _tokenPrice;
        uint exchange = weis % _tokenPrice;
        uint oldExchange = participantExchange[_participant];

        //Comprobar con el antiguo cambio si puede recibir un token m s
        if(exchange + oldExchange >= _tokenPrice){
            tokens += 1;
            unchecked{ //Por el if
                participantExchange[_participant] = exchange + oldExchange - _tokenPrice;
            }
        }
        else{
            unchecked{
                participantExchange[_participant] = exchange + oldExchange;
            }
        }

        return tokens;
    }
}
```

2. Detalles de implementación

2.1. Quadratic Voting

Contrato encargado de controlar toda la Dapp. Sigue todas las especificaciones marcadas por el enunciado. En cuanto a las libertades de diseño:

TotalBudget

variable de estado que almacena el presupuesto actual disponible del contrato para las propuestas. No es exactamente el balance del contrato, pues este se actualiza cuando un usuario compra tokens, y no cuando una propuesta ha sido aprobada. Por lo tanto, totalBudget solo se actualiza en la función *_checkAndExecuteProposal()*, una vez se ha aprobado una propuesta.

Listas de proposals

Las propuestas se almacenan en 3 listas diferentes según su tipo: **signaling**, **financialNotApproved** y **financialApproved**. Cada una se actualiza en directo, es decir, si una propuesta se cancela desaparece de la lista, o bien si es financial y se aprueba pasa a la lista de aprobadas. Otras opciones de diseño eran llevar una única lista y el número de cada tipo o únicamente separar signaling de financial. En ambos casos, para los gets de los tipos de propuestas el coste es muy superior, pues hay que recorrer al menos una lista. Además, si se desea cancelar una, se necesitan los mismos accesos a storage. La única diferencia sería en caso de aprobar, que tan solo en esos casos se modificaría una variable del struct que explicaremos a continuación.

Por lo tanto, como creemos que los gets de las propuestas se van a ejecutar como mínimo el mismo número de veces que votar (cada usuario una vez votado quiere saber si se ha aprobado o no su propuesta, además de todos aquellos que simplemente actúan por curiosidad), consideramos que la mejor opción es implementar las 3 listas por separado.

Dada la posición de una propuesta que debe abandonar una lista, para borrarla de ella, intercambiamos dicha posición con la última del array. Acto seguido hacemos un pop para expulsarla. No importa que el array esté desordenado.

Mappings

Participants: sirve para ver si un usuario ya es participante de la Dapp. Una vez alguien se convierte en participante, nunca deja de serlo. Esto significa que los participantes son independientes de los periodos de votaciones. También utilizamos un array de los participantes para poder devolverles los tokens y controlar el número de participantes. Aunque el mapping sea redundante, permite comprobar si una cuenta es participante sin necesidad de recorrer el array. Así conseguimos abaratar de forma notoria el coste.

ProposalsInfo: asocia el id de una propuesta con su información, que se almacena en el struct ProposalInfo, que se explica a continuación. El id es un número entero inicialmente a 1 que aumenta según se crea una propuesta. Es una forma de tener, además, el número histórico de propuestas y que el id de estas sea siempre único.

ParticipantVotes: para cada participante y para cada propuesta se almacenan los votos que ha depositado. Se utiliza para devolver los tokens de una propuesta cancelada a los participantes. Otra opción era llevar para cada propuesta en su struct dicha información, pero significa tener mucha información repetida, lo que se traduce en más storage ocupado. Además, sería igualmente necesario iterar sobre todos los participantes, pues no se puede iterar sobre el mapping directamente.

ParticipantExchange: este mapping asocia a cada cuenta de participante el cambio en ether que tiene a la hora de comprar tokens o a la hora de inscribirse como participante (aunque en este caso sí obligamos a comprar al menos 1 token). Llevamos esto así porque nosotros no obligamos a comprar una cantidad fija de ether ni queremos revertirle la compra o transferirle ethers si esta no es exacta. Entonces, cuando ellos

compran tokens si la compra no es exacta se les almacena el cambios devuelto, que se le queda almacenado para una compra posterior, de forma que si cuando compras el cambio que sobra más el que tenía ya almacenado da para un token más se le transfiere este token y se le actualiza el cambio. Debido a esto, el uint guardado en este mapping siempre es ¡1. El participante siempre puede consultar el cambio que posee con la función *getExchange* que implementamos y también puede recuperar este cambio si lo desea con la función *regainExchange*.

ProposalInfo

Struct que almacena toda la información de una propuesta: tittle, description, budget (para saber además, si una propuesta es signaling cuando este es 0)... Además, almacenamos ciertos atributos necesarios para nuestra implementación.

En *indexList* almacenamos el índice en el que está la propuesta en su correspondiente lista de propuestas para a la hora de cancelar o aprobar una propuesta no tener que recorrer todo el array para encontrarla sino ya saber donde está. Así, cuando vamos a eliminar una propuesta de una lista la sustituimos por el último elemento y, posteriormente, hacer pop para poder eliminarla fácilmente del array.

En *isApproved* controlamos si una propuesta ha sido aprobada o no. Aunque esta info es rebundante, es útil para los modifiers de ver si una propuesta ha sido o no aprobada sin tener que recorrernos las listas, siendo así más eficiente.

Votes y tokens guardan la información del nº de votos y tokens recibidos por los participantes en esa propuesta. Los guardamos en el struct porque a la hora de comprobar y/o ejecutar una propuesta necesitamos saber estos datos y lo más eficiente es guardarlos en este struct antes que en un mapping aparte o tener que acceder al mapping participantVotes para calcular estos datos de la propuesta según votos (y tokens) de cada participante.

Locks

Variable booleana que permite bloquear la llamada a la función ExecuteProposal de un contrato propuesta externo. Como no nos debemos fiar de las personas que implementan dichos contratos, esta llamada requiere protección. En caso contrario podríamos sufrir el ataque *Reentrancy Attack* y el contrato maligno podría robar todo el balance de QuadraticVoting, destruyendo así la Dapp.

Se incluye en la función _checkAndExecuteProposal(), pero en closeVoting no hace falta porque solo la puede ejecutar el admin. En todo caso, siempre actualizamos antes de la llamada a ExecuteProposal el estado: los votos se ponen a 0 antes de ejecutar una Signaling para que no se puedan retirar; y la propuesta Financial se marca como aprobada para que no se pueda volver a votar y recibir el budget de nuevo o retirar votos de ella. Además, en closeVoting primero ponemos isOpen a false para que el contrato externo no pueda ejecutar nada.

CalculateTokens

Función interna que dada una cantidad de weis y un pagador (ha de ser participante), calcula el número de tokens que puede comprar. Tiene en cuenta y actualiza el cambio que le queda a dicho participante.

Versión del compilador

Utilizamos la versión 0.8.7. No tenemos problemas de tamaño del código compilado. Tampoco utilizamos SafeMath porque esta versión se encarga de todas las comprobaciones. Como algunas instrucciones son seguras de ejecutar, por ejemplo solo se llegan a ellas tras require, introducimos unchecked para que el compilador tenga menos trabajo.

2.2. TokenManager

Contrato encargado de controlar la lógica de los tokens ERC-20. Para ello, hereda del contrato ERC2O, implementado por la organización *OpenZeppelin*. A parte de las funciones que ya vienen desarrolladas, es necesario incluir **mint()** y **burn()**, para crear y destruir tokens respectivamente. Además, almacena la dirección del contrato QuadraticVoting, pues es la única que debe ejecutar estas 2 nuevas funciones. Los usuarios tan solo pueden transferir tokens entre ellos, no deben crearse nuevos ni destruirlos.

El mint y el burn se utilizan para modificar la cantidad de tokens de la Dapp. Para ejecutar el mint, siempre es necesario tener en cuenta que no se puede superar la cantidad máxima de tokens *maxTokens*. Otra opción de diseño es hacer un mint del máximo de tokens a la cuenta de QuadraticVoting, y a partir de ahí en vez de emplear el mint, se utiliza el transfer para darle a los participantes los tokens que van comprando. Sin embargo, como ya se ha implementado mint, consideramos que es más natural llamarla cada vez que se creen nuevos tokens.

Cuando un usuario quiere comprar tokens, pero con el dinero aportado se superarían el máximo de tokens, revertimos la operación. Consideramos que un usuario puede haber introducido una cantidad para comprar exactamente esos tokens y puede no querer menos. En dicho caso, tras revertirse la operación, el usuario puede ver cuántos quedan disponibles.

3. Prueba de ejecución

Primero probamos todas las funciones por separado para comprobar que su funcionalidad era la esperada. Acto seguido, elaboramos un guión de prueba:

1. **Cuenta1** despliega el contrato QuadraticVoting. Será el administrador, con permisos únicos. Cada token va a costar 2 weis y el máximo va a ser 1000.
2. **Cuenta2** se añade como participante. Paga 21 weis. obtiene 10 tokens y de cambio tiene un wei.
3. **Cuenta2** intenta añadir una propuesta. No puede porque la votación no está abierta.
4. **Cuenta2** intenta abrir el periodo de votación. No puede porque no es admin.
5. **Cuenta1** abre el periodo de votación. Se le da un presupuesto de 1000 weis.
6. **Cuenta3** intenta añadir una propuesta. No puede porque no es participante.
7. **Cuenta3** se añade como participante. Paga 30 weis y recibe 15 tokens, con cambio de 0 weis.
8. **Cuenta4** se añade como participante. Paga 7 weis y recibe 3 tokens, con cambio de 1 wei.
9. **Cuenta4** comprueba su cambio. Da 1 wei.
10. **CuentaX** ejecuta getRestTokens y quedan $972 = 1000 - (10 + 15 + 3)$.
11. **CuentaX** despliega 2 contratos de *finacial* proposal y 1 de *signaling*.
12. **Cuenta2** añade una propuesta *financial* con un budget de 20 weis.
13. **Cuenta2** añade una propuesta de *signaling*. Tiene budget 0, si no no sería signaling.
14. **Cuenta3** añade otra propuesta de financial con un budget de 50 weis.
15. **CuentaX** comprueba que las estructuras de las propuestas son correctas.
16. **Cuenta4** intenta cancelar la propuesta con id 3. No puede porque no es el creador.
17. **Cuenta4** intenta poner 2 votos en la propuesta 3. No puede porque no tiene suficientes tokens.
18. **Cuenta4** paga 1 wei para comprar otro token con el cambio. Ahora tiene 4 tokens y un cambio de 0 weis.
19. Para ver esto último, **CuentaX** ejecuta la función getBalance de la instancia de TokenManager del QuadraticVoting.
20. **Cuenta4** aprueba al contrato QuadraticVoting para gastar los 4 tokens.
21. **Cuenta4** pone 2 votos en la propuesta 3. Pasa a tener 0 tokens disponibles.
22. **Cuenta4** comprueba que los votos de la propuesta se han actualizado con getProposalInfo.
23. **Cuenta3** cancela la propuesta 3. Cuenta4 recibe de vuelta los 4 tokens asociados a los 2 votos que había puesto.
24. **CuentaX** comprueba que la propuesta no existe, ejecutando getProposalInfo.
25. **Cuenta4** aprueba al contrato QuadraticVoting para gastar los 4 tokens.
26. **Cuenta4** pone 2 votos en la propuesta 2 (signaling). Pasa a tener 0 tokens disponibles de nuevo.
27. **Cuenta2** aprueba al contrato QuadraticVoting para gastar 4 tokens.
28. **Cuenta2** pone 2 votos en la propuesta 1 (financial). Pasa a tener $10 - 4 = 6$ tokens disponibles.

29. **Cuenta2** intenta poner otros 2 votos en la propuesta 1 (financial). No puede porque no tiene suficientes (necesitaría 12 y tiene 6). Pone 1 voto. Le queda 1 token disponible.

30. La propuesta 2 se aprueba acto seguido. Las condiciones son: $1000 + 2 * 9 > 20$ y $10 * 1000 * 3 = 30000 > (2 * 1000 + 10 * 20) * 3 + 10 * 1000 * 2 = 26000$. Se envía un evento que lo confirma. TotalBudget pasa a ser 998.

31. El contrato de la financial proposal emite un evento al ser aprobada, que nos permite comprobar que todo funciona correctamente. Ha recibido 20 weis (su budget) y los votos y tokens son correctos.

32. **CuentaX** comprueba que se han quemado los tokens con getRestTokens. Ahora quedan 980 disponibles.

33. **Cuenta1** cierra la votación. Se emite otro evento Executing, de la propuesta signaling. Con balanceOf comprobamos que **Cuenta4** tiene los 4 tokens de vuelta.

El contrato pasó todas estas fases de pruebas.