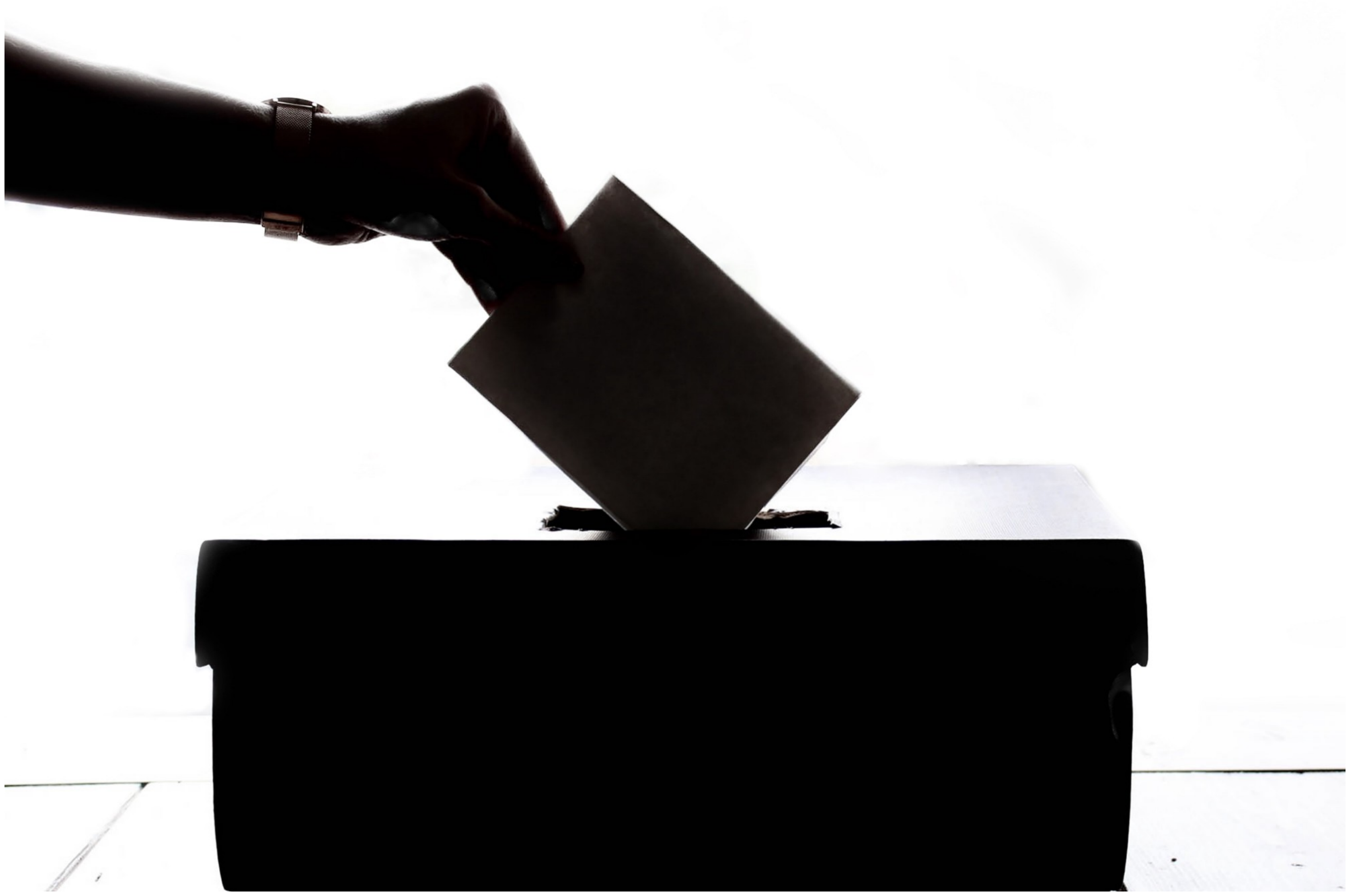
How to build a distributed token-based voting app on Ethereum

Doug Crescenzi Follow
Feb 20 · 6 min read ★



There are many different ways to build distributed governance systems on Ethereum. One of the most common approaches is for constituents to use tokens to represent their votes. The more tokens you have, the more votes you're able to cast.

Not all distributed token-based voting applications are the same, but most follow similar conventions. The workflow generally looks something like this:

> A constituent submits a proposal
>
> Other constituents can then vote in favor of or against the proposal
>
> Once the proposal reaches a predefined threshold, the vote passes or fails and appropriate corresponding actions are taken

Sounds easy enough, right?

There are however challenges that come with implementing distributed token-based voting apps. Voter fraud is at the top of the list. There are also challenges with how state is managed and how unique characteristics associated with certain tokens should work in conjunction with corresponding governance mechanisms.

With that said, let’s take a look at how you might build a distributed token-based voting app on Ethereum. To do so, let’s assume you plan to build two contracts that interact with each other:

> A Token contract that contains a mapping of account addresses and their balances
>
> A Governance contract that is used to manage the voting system

For this example constituents will vote on proposals using their token balances. Their token balances will represent the number of votes they can cast on a particular proposal.

A constituent submits a proposal

First things first, we need to develop a function that will allow a constituent to submit a proposal to our Governance contract.

```
// Map a proposal ID to a specific proposal
mapping(uint => Proposal) public proposals;

// Map a proposal ID to a voter's address and their vote
mapping(uint => mapping(address => bool)) public voted;
```

```
  // Determine if the user is blocked from voting
  mapping (address => uint) public blocked;


  struct Proposal {
    uint votesReceived;
    bool passed;
    address submitter;
    uint votingDeadline;
  }


/// @dev Allows a token holder to submit a proposal to vote
on
  function submitProposal()
    public
    onlyEligibleVoter(msg.sender)
    whenNotBlocked(msg.sender)
    returns (uint proposalID)
  {
    votesReceived = token.balanceOf(msg.sender);
    proposalID = addProposal(votesReceived);
    emit ProposalSubmitted(proposalID);
    return proposalID;
  }
```

Before allowing a constituent to submit a proposal, we need to verify that they are eligible to vote. In our case we need to make sure a constituent has a token balance that is greater than 0. We'll use our `onlyEligibleVoter` modifier to check and see if a constituent is eligible:

```
modifier onlyEligibleVoter(address _voter) {
   balance = token.balanceOf(_voter);
   require(balance > 0);
  _;
}
```

Once we've determined that a voter is eligible, the `submitProposal` function will assign the submitters token balance to a variable `votesReceived` . We'll then pass this value as an argument to our `addProposal` function:

```
/// @dev Adds a new proposal to the proposal mapping
/// @param _votesReceived from the user submitting the
proposal

  function addProposal(uint _votesReceived)
   internal
   returns (uint proposalID)
  {
   votes = _votesReceived;
   if (votes < votesNeeded) {
      if (proposalIDcount == 0) {
        proposalIDcount = 1;
      }
    proposalID = proposalIDcount;
    proposals[proposalID] = Proposal({
    votesReceived: votes,
    passed: false,
    submitter: msg.sender,
    votingDeadline: now + voteLength
     });
    blocked[msg.sender] = proposalID;
    voted[proposalID][msg.sender] = true;
    proposalIDcount = proposalIDcount.add(1);
    return proposalID;
   }
   else {
    require(token.balanceOf(msg.sender) >= votesNeeded);
    endVote(proposalID);
    return proposalID;
   }
  }
```

The `addProposal` function will generate a unique ID for the proposal. It will also create a `Proposal` object that can be used to monitor how many votes the proposal has received, whether or not it has passed, and the voting deadline for the proposal.

After the proposal has been added, the `submitProposal` function will return the proposal's unique ID and emit a `ProposalSubmitted` event for our frontend to consume.

Other constituents can vote in favor of or against the proposal

Next, we need to develop a function to process votes from constituents who would like to vote in favor of a particular proposal.

In this function we first have an `if` statement that we use to determine if a particular voter is blocked from voting (more on this later). If a voter is not blocked, we determine what their token balance is and add that value to the specific proposal's `votesReceived` variable.

```
/// @dev Allows token holders to submit their votes in favor
of a specific proposalID
/// @param _proposalID The proposal ID the token holder is
voting on

  function submitVote(uint _proposalID)
    onlyEligibleVoter(msg.sender)
    whenNotBlocked(msg.sender)
    public
    returns (bool)
  {
    Proposal memory p = proposals[_proposalID];


    if (blocked[msg.sender] == 0) {
      blocked[msg.sender] = _proposalID;
    } else if (p.votingDeadline >
proposals[blocked[msg.sender]].votingDeadline)
    {


// this proposal's voting deadline is further into the
future than
// the proposal that blocks the sender, so make it the
blocker


      blocked[msg.sender] = _proposalID;
    }
    votesReceived = token.balanceOf(msg.sender);
    proposals[_proposalID].votesReceived += votesReceived;
    voted[_proposalID][msg.sender] = true;
    if (proposals[_proposalID].votesReceived >= votesNeeded)
    {
      proposals[_proposalID].passed = true;
      emit VotesSubmitted(
        _proposalID,
        votesReceived,
        proposals[_proposalID].passed
      );
      endVote(_proposalID);
    }
    emit VotesSubmitted(
      _proposalID,
      votesReceived,
      proposals[_proposalID].passed
    );
    return true;
  }
```

We then check to see if the proposal has passed as a consequence of the new votes it has received. If it has, we'll emit a `VotesSubmitted` event and call our `endVote` function. If it has not passed, we'll simply emit a `VotesSubmitted` event and return `true` to acknowledge that the votes were received and processed successfully.

The vote passes and appropriate corresponding actions are taken

Not that we've developed a function that allows constituents to create proposals and submit votes, let's take a look at what happens after a proposal receives enough votes to pass.

In our `submitVote` function you'll recall this `if` statement that checks to see if enough votes have been received for a proposal to pass:

```
if (proposals[_proposalID].votesReceived >= votesNeeded) {
    proposals[_proposalID].passed = true;
    emit VotesSubmitted(
       _proposalID,
       votesReceived,
       proposals[_proposalID].passed
    );
    endVote(_proposalID);
   }
```

If it has received enough votes, the `endVote` function is called which looks like this:

```
/// @dev Sets when a particular vote will end
/// @param _proposalID The specific proposal's ID

  function endVote(uint _proposalID)
    internal
  {
    require(voteSuccessOrFail(_proposalID));
    updateProposalToPassed(_proposalID);
  }
```

You'll notice we're using a `require` statement to ensure a particular proposal has received enough votes to pass in conjunction with our `voteSuccessOrFail` function:

```
/// @dev Determines whether or not a particular vote has
passed or failed
/// @param _proposalID The proposal ID to check
/// @return Returns whether or not a particular vote has
passed or failed

  function voteSuccessOrFail(uint _proposalID)
    public
    view
    returns (bool)
  {
    return proposals[_proposalID].passed;
  }
```

If the vote has passed, we'll call a function that updates the `Proposal` 's `passed` variable to `true` .

How to prevent voter fraud

The biggest challenge with distributed token-based voting apps is the potential for voting fraud. For instance, what would stop a constituent from voting on a proposal and then sending their tokens to a different wallet and re-voting with the same tokens that they've already voted with?

If you'll recall from earlier we encountered a mapping named `blocked` .

```
// Determine if the user is blocked from voting
  mapping (address => uint) public blocked;
```

We're using this mapping to determine whether or not a particular constituent is `blocked` from voting. We block a constituent from voting if they've already voted or submitted a proposal, and the proposal's `votingDeadline` has not yet elapsed. If a users is `blocked` they are unable to transfer their tokens until the `votingDeadline` has elapsed.

To implement this in our token contract we'll use the following `whenNotBlocked` modifier:

```
/// @dev Modifier to check if a user account is blocked from
making transfers

    modifier whenNotBlocked(address _account) {
      require(!governance.isBlocked(_account));
      _;
    }
```

We'll then use the modifier in our token contract for `transfer` and `transferFrom` which inherits from OpenZeppelin's ERC-20 token contract.

```
 function transfer(address to, uint256 value)
      public
      whenNotBlocked(msg.sender)
      returns (bool)
    {
      return super.transfer(to, value);
    }

function transferFrom(address from, address to, uint256
value)
      public
      whenNotBlocked(from)
      returns (bool)
    {
      return super.transferFrom(from, to, value);
    }
```

A shortcoming to this approach is that it disincentives voting as constituents will be unable to transfer their tokens if they're in a `blocked` state. Given that, it's important to have a solid grasp on your token's underlying utility and the impact this approach may have on your users. This is a great article that describes other ways to vote safely with ERC-20 tokens.

Takeaways

> There's not a single "correct" way to build a token-based voting application. It will depend on the use case.
>
> Most token-based voting applications follow a pattern like this: A constituent submits a proposal → other constituents can then vote in favor of or against the proposal → once the proposal reaches a predefined threshold, the vote passes or fails and appropriate corresponding actions are taken.
>
> Voter fraud is a big challenge to address when it comes to implementing distributed token-based voting apps. The fungibility of ERC-20 tokens makes it hard to prevent this.
>
> There are also challenges to consider associated with how state is managed. I.e., how should unique characteristics associated with certain tokens work in tandem with corresponding governance mechanisms
>
> Ultimately, token-based voting is a powerful governance mechanism for disturbed applications, but they introduce risks that require mitigation.

Interested in learning more about Ethereum development?