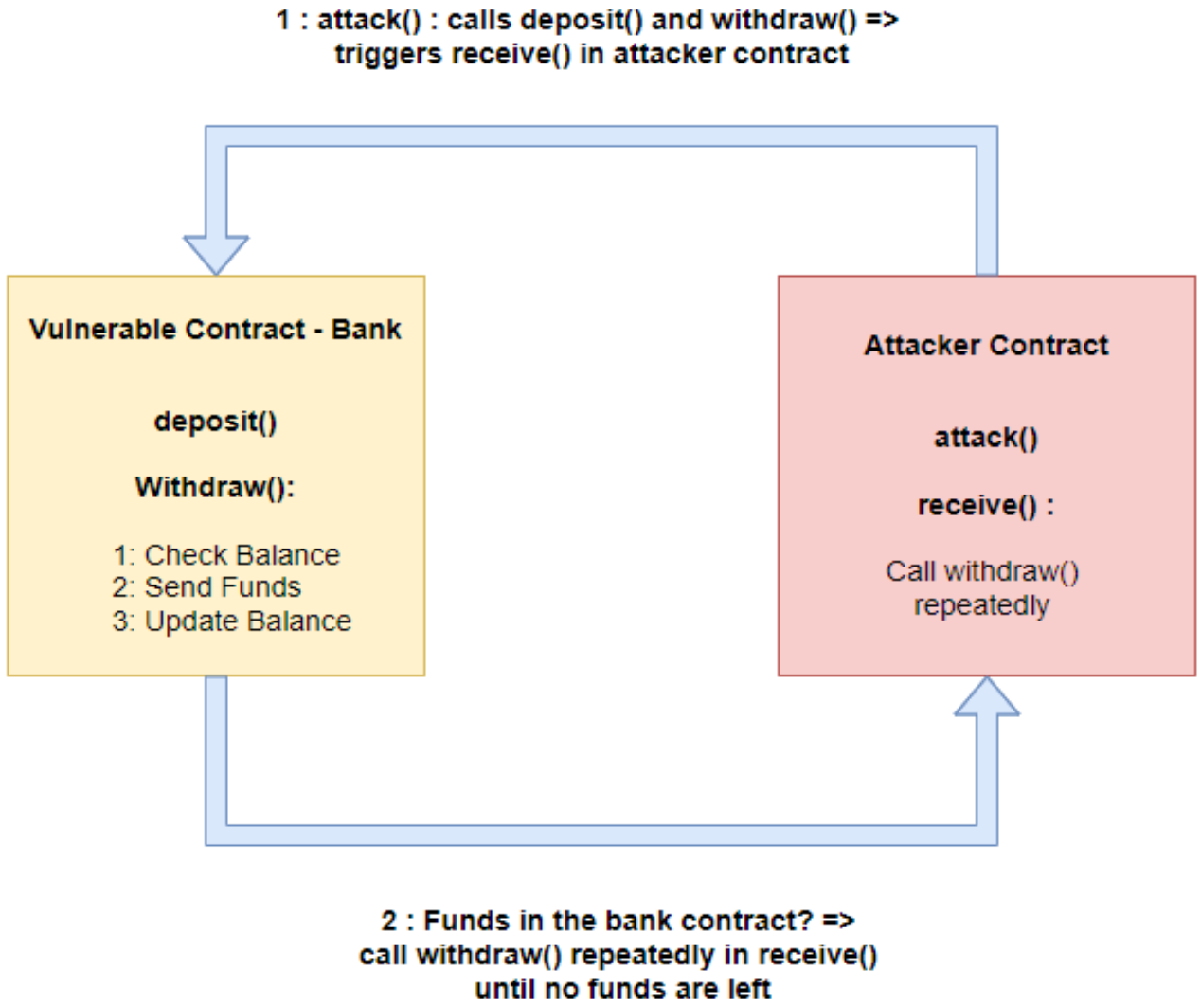
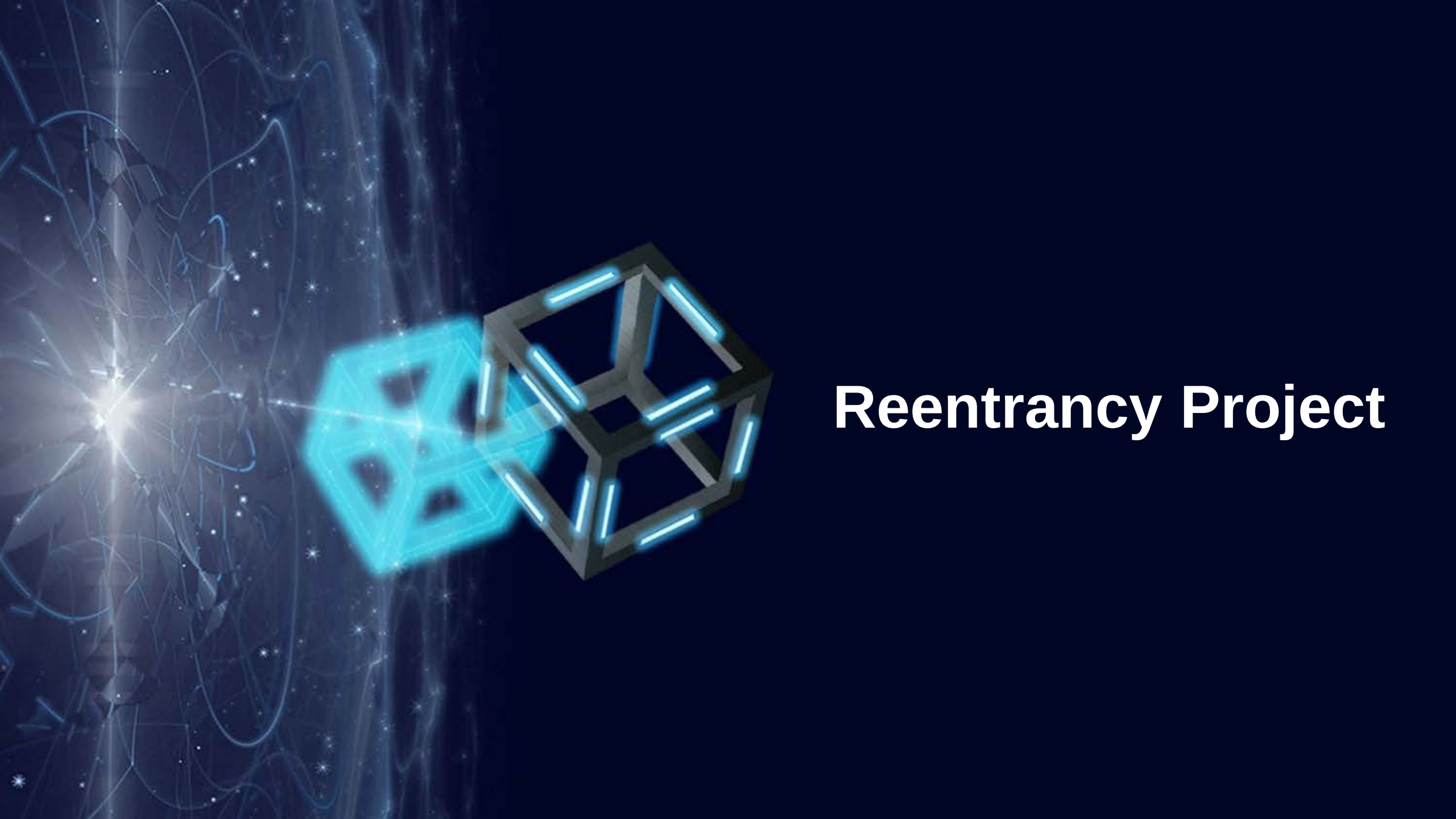

ETHEREUM SMART
CONTRACT DEVELOPMENT
Reentrancy - DAO

Reentrancy Attack

Reentrancy Attack



- ATTACKER deposits funds to BANK and then calls withdraw()
- BANK checks if the ATTACKER has the funds, then it transfers the funds to ATTACKER
- When the ATTACKER receives the funds, it executes a fallback function which calls again withdraw() on BANK before it is able to update its balance
- This process continues until all the funds have been drained from BANK



Reentrancy Project

Reentrancy Project - Requirements

Bank Contract:

- Manage balance of users
- Deposit funds
- Withdraw funds
  - Deposited amount must be > 0
  - Transfer funds to user
  - Update user balance in contract

Attacker Contract:

- Create Bank contract in constructor
- Attack Bank contract: call ***deposit() and withdraw()***
- ***receive():*** check if funds left in Bank contract => call withdraw

Protecting Against a Reentrancy Attack

- Bad design of vulnerable contract: check balance => send funds => update balance
- Time between sending the funds and updating the balance creates a window in which the attacker can make another call to withdraw funds => the cycle continues until all the funds are drained
- Checks-Effects-Interaction pattern – recommended guideline for coding smart contracts:
  - 1: validate input data (check balance)
  - 2: make changes to state variables (update balance)
  - 3: interact with other contracts (send funds)
- Ethereum DAO hack 2016: 3.6 M ETH ($60 million dollars) were stolen => To regain those funds, a hard fork was initiated. The original blockchain is now Ethereum Classic

Reentrancy Project - Exercise

Protecting the Bank contract:

- Inherit Bank contract from ***ReentrancyGuard*** (OpenZeppelin)
- Add ***nonReentrant*** modifier to withdraw function
- Apply the **Check-Effects-Interaction** pattern
- By using transfer instead of call this attack would not have been possible. However, the transfer method limits gas consumption in the receive function to 2300 units of gas => this could break some existing contracts, because the gas cost may change for various opcodes, therefore it is no longer recommended to use the transfer method.

DAO

Decentralized Autonomous Organization

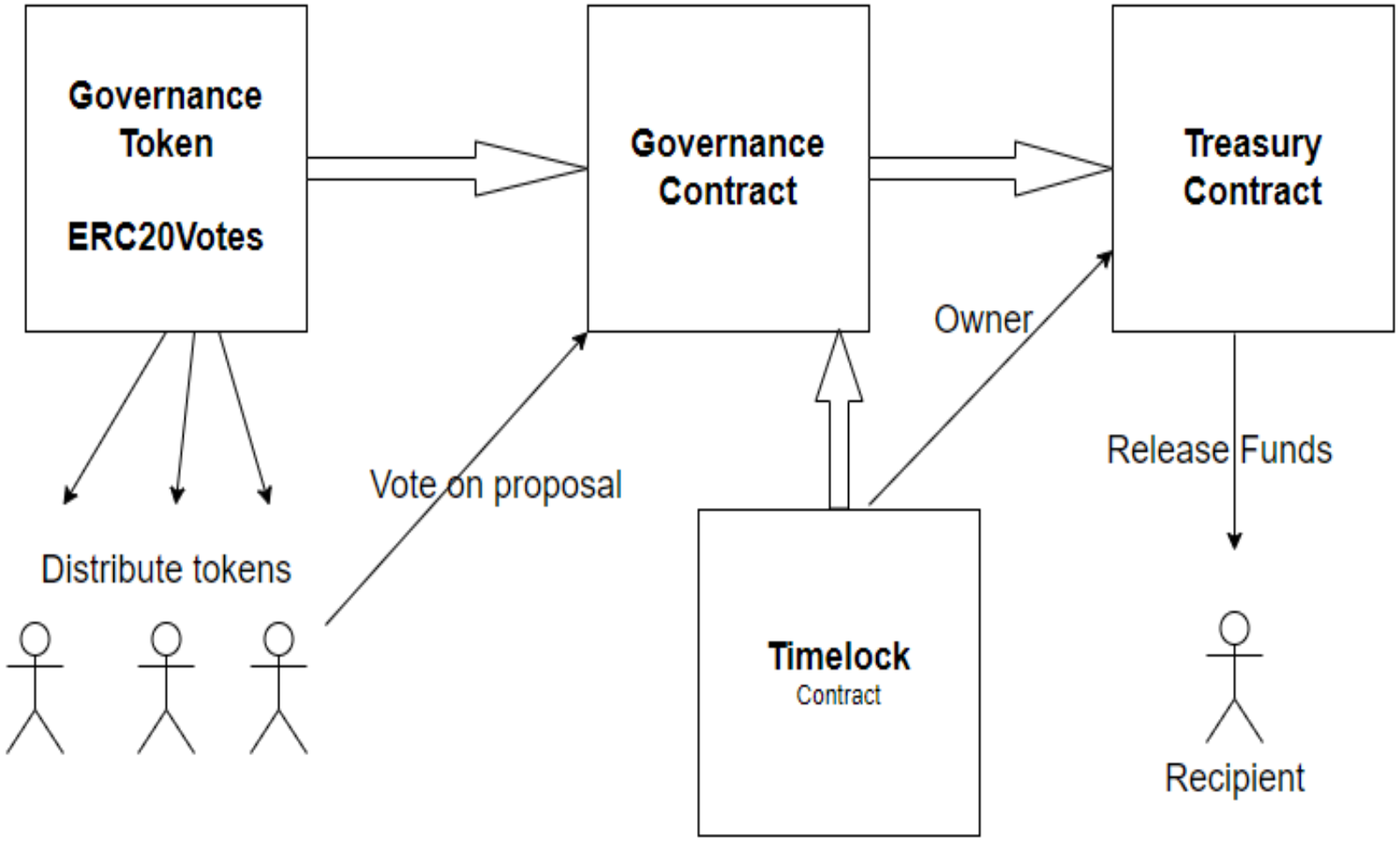
What is a DAO?

- Community-led entity with no central authority
- Fully autonomous and transparent: smart contracts execute the agreed upon decisions, and at any point, proposals, voting, and even the smart contract code can be publicly audited
- Governed entirely by its individual members who collectively make critical decisions
- The rules of the DAO are established by a core team of community members and they are enforced by smart contracts
- Community members create proposals about various operations and then they vote on each proposal
- Proposals that achieve some predefined level of consensus are then accepted and enforced by smart contracts

How Does a DAO Work?

- Once the rules are deployed to the blockchain (via smart contracts), the DAO needs to figure out how to receive funding
- Typically achieved through token issuance, by which the protocol sells tokens to raise funds and fill the DAO treasury
- Token holders are given voting rights, usually proportional to their holdings
- Once funding is completed, the DAO can become operational - proposals can be created, members can vote and successful proposals can be executed
- In certain protocols such as Compound, token holders can vote to distribute treasury funds towards bug fixes and system upgrades. This approach also allows developers to join ad hoc and receive compensation for their work

Project Contracts



- Distribute governance tokens - via crowd sale or other means
- Create a proposal on the governance contract
- Governance token holders can vote on that proposal
- If the vote is successful, the proposal is executed and the funds are sent to the recipient
- The timelock enforces a delay after which the funds are released

Governance Token

ERC20Vote:

- Extension of ERC20 to support voting and delegation
- Keeps a history (checkpoints) of each account’s vote power

Timelock Contract - TimelockController

- The Treasury contract holds the funds for the proposal.
- When the vote passes, the Timelock contract, which is the owner of the Treasury contract executes the proposal (after a specific delay) and transfers those funds to the receiver

AccessControl of TimelockController:

- The Proposer role (granted to Governor) is in charge of queueing operations
- The Executor role (can also be granted to Governor) is in charge of executing already available operations
- The Admin role (granted automatically to timelock) can grant and revoke the two previous roles

Operations / Transactions

An operation is a transaction (or a set of transactions) that can be created by any member of the community (proposal). At the end of the voting period it has to be scheduled (queued) by a Proposer and executed (after a minimum delay) by an Executor.

Operations are identified by a unique id and follow a specific lifecycle:

- **Pending**: after the creation of a proposal
- **Active**: voting ongoing - voting period has not passed yet
- **Queued**: after the operation has been scheduled (queued)
- **Executed**: after the operation has been executed

Proposer calls "***queue***" => moves operation from ***Active*** to ***Queued*** => timer is started => once timer expires, the Executor can execute the operation (send transaction(s)) => operation moves to ***Executed*** state

Creating a Proposal

A proposal is a sequence of action(s) that the Governor contract will execute if it passes. Each action consists of a target address, calldata (encoded function call), the amount of ETH to include and a description.

Executing the proposal:

- Call the ***propose*** function of the governor with the required parameters:
  await governor.propose([contractAddress], [0], [transferCalldata], "Give grant to team")
- Delegates can cast their vote
- Once the voting period is over, if quorum was reached, the proposal is considered successful and can queued for execution => Call the ***queue*** function of the governor
- After the required time delay (if there is one) the proposal can be executed => call the ***execute*** function of the governor

Governance Contract – Required Modules

- ***Governor***: Contains the core logic. We can provide a name for the DAO in the constructor.
- ***GovernorVotes***: Determines the voting power of an account based on its governance token balance. We need to provide address of ***governance token*** in constructor.
- ***GovernorVotesQuorumFraction***: Defines quorum as a percentage of the total token supply. We need to provide the desired ***quorum*** in the constructor. Most governors require a quorum of 4-5%.
- ***GovernorCountingSimple***: Offers 3 options to voters: For, Against, and Abstain. Only For and Abstain votes are counted towards quorum.
- ***GovernorTimelockControl***: Connects with an instance of a TimelockController. We need to provide an instance of a ***TimelockController*** in the constructor.

Governance Contract – Required Parameters

We also need to set the following parameters:

- ***votingDelay***: Delay (in number of blocks) from the creation of a proposal until voting starts.
- ***votingPeriod***: Delay (in number of blocks) from the creation of a proposal until voting ends.
- ***quorum***: Quorum required for a proposal to be successful

Governance Contract – Interface

- ***propose***(address[] targets, uint256[] values, bytes[] calldatas, string description) → uint256 proposalId // create a new proposal
- ***state***(uint256 proposalId) → enum IGovernor.ProposalState // return current proposal state
- ***castVote***(uint256 proposalId, uint8 support) → uint256 balance // cast a vote
- ***proposalVotes***(uint256 proposalId) → uint256 againstVotes, uint256 forVotes, uint256 abstainVotes // get the vote distribution
- ***queue***(address[] targets, uint256[] values, bytes[] calldatas, bytes32 descriptionHash) → uint256 // queue a proposal
- ***execute***(address[] targets, uint256[] values, bytes[] calldatas, bytes32 descriptionHash) → uint256 proposalId // execute a successful proposal

DAO Project

DAO Project - Requirements

Create the required contracts:

- ***GovToken:*** ERC20Votes - provide name, symbol and initial supply => mint initial supply to token owner address
- ***Treasury:*** Ownable contract, holds the funds, allow transfer of funds to payees
- ***Timelock:*** TimelockController - provide minDelay, proposer(s) and executor(s)
- ***Governance:*** Governor, GovernorCountingSimple, GovernorVotes, GovernorVotesQuorumFraction, GovernorTimelockControl
  => provide voting delay and voting period

DAO Project - Requirements

Create the deployment script:

- Signers for executor, proposer, payee and 5 voters
- Deploy **Governance token**: initial supply: 1000
- Transfer 50 governance tokens to each voter
- Deploy **Timelock** contract: minDelay = 0
- Deploy **Governance** contract: quorum = 5, votingDelay = 0, votingPeriod = 5
- Timelock grants proposer and executor roles to the governance contract
- Deploy **Treasury** contract: transfer funds (50 ETH),
  transfer ownership to **Timelock**

DAO Project - Requirements

Create the proposal script 1/2:

- Self-delegate voting rights to 5 voters
- Display initial treasury and payee balance
- Encode the ***releaseFunds*** function from the treasury contract => required for the proposal
- Create a proposal: There is one action => target: address of treasury, value: 0, calldata: encoded "**releaseFunds**" function
- Display the proposal Id from the ***ProposalCreated*** event
- Display the state of the proposal

DAO Project - Requirements

Create the proposal script 2/2:

- Display the blocks for the proposal creation and the proposal deadline
- Cast the votes
- Display the number of votes (for, against and abstain)
- Queue the proposal
- Execute the proposal
- Display the state of the proposal
- Display final treasury and payee balance