НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені Ігоря СІКОРСЬКОГО»
Навчально-науковий фізико-технічний інститут
Кафедра математичних методів захисту інформації

Звіт до лабораторної №2

за темою:
Реалізація смарт-контракту
та/або анонімної криптовалюти

Оформлення звіту:
Юрчук Олексій, ФІ-52мн

22 лютого 2026 р.
м. Київ

ЗМІСТ

1 Вступ

За мету в цій лабораторній роботі я ставив собі попрацювати і набути практичних навичок зі смарт-контрактами Ethereum (у продовження першої лаби). У Завданні 1 я візьму один з еталонних контрактів з офіційної документації Solidity (версія 0.5.3, «Solidity by Example»), розгорну його в тестовій мережі Sepolia та спробую покращити його газову ефективність. А потім ще спробую накняпати ще контракт власними силами (а-ля завдання 2).

На сайті Solidity наведено такі приклади контрактів:

1. **Voting (Ballot)** — система делегованого голосування з контролем доступу.
2. **Simple Auction** — відкритий аукціон з ефірним депозитом.
3. **Blind Auction** — аукціон з розкриттям ставок після їхнього підтвердження з хешованими ставками.
4. **Payment Channel** — позаланцюгові мікроплатежі з підписами.

Зупинюся я на контракті **Ballot (Voting)** (Голосування), оскільки він демонструє найширший спектр можливостей мови Solidity – структури, відображення, контроль доступу, логіку делегування та функції перегляду, і при цьому не вимагає переказів Ether, що спрощує тестування. Крім того, його механізм делегування містить чітку можливість оптимізації газу (необмежений цикл `while`).

2 Як працюють смарт-контракти (in general)

2.1 Визначення

Smart contract це певна програма, що зберігається в блокчейні Ethereum і виконується автоматично при виклику її функцій. Життєвий цикл складається з чотирьох етапів:

1. **Writing.** Розробник пише вихідний код у Solidity (`.sol`). Компілятор (`solc`) створює два артефакти:
   - *EVM bytecode* — низькорівневі інструкції, що виконуються віртуальною машиною Ethereum.
   - *ABI (Application Binary Interface)* — опис у форматі JSON усіх публічних функцій, їхніх параметрів та типів повернення. Зовнішні інструменти, такі як (web3.js, ethers.js, Remix), використовують ABI для кодування/декодування викликів.
2. **Deployment.** Спеціальна транзакція з порожнім полем `to` надсилає байт-код до мережі. EVM виконує `constructor()` рівно один раз, ініціалізує змінні стану, а мережа присвоює контракту постійну адресу.
3. **Interaction.** Користувачі (або інші контракти) викликають функції одним із двох способів:
   - *State-changing functions* (write) — створюють транзакцію, яка має бути підтверджена в блокчейні; (cost gas).
   - *View / pure functions* (read) — виконується локально by node, (free of charge).
4. **Termination.** Контракт існує в ланцюжку безстроково. Його можна зробити нефункціональним за допомогою операційного коду `selfdestruct` (застарілого в сучасній Solidity) або шляхом впровадження шаблону паузи/знищення.

2.2 Стан, гас і вартість контрактів (EVM Storage and Gas)

Стан контракту зберігається в постійній пам'яті (*storage slots*), кожен з яких має ширину 256 біт. Запис у новий слот коштує 20000 gas (`SSTORE`); оновлення існуючого слота коштуватиме 5 000 гасу; читання коштує 2 100 гасу (`SLOAD`). Пам'ять (тимчасова, для кожного виклику) і дані виклику (тільки для читання, для аргументів функції) значно дешевші.

Кожен виконуваний EVM код має фіксовану гасову вартість. Відправник транзакції вказує *gas limit*, а також сплачує gas used × gas price в ETH. Якщо гас закінчується в процесі виконання, всі зміни стану скасовуються, і (що важливо!) спожитий гас на рахунок *не* повертається. Цей механізм запобігає нескінченним циклам і стимулює писати ефективний код.

2.3 Події (Events)

Контракти можуть `emit` (продукувати) певні події. Ці події зберігаються в ланцюжку як журнали транзакцій (це набагато дешевше, ніж зберігання: ∼375 gas base + 375 gas per indexed topic + 8 gas per byte of data). Вони доступні для запиту поза ланцюжком, але *не* доступні для читання іншими контрактами.

2.4 Діаграма життєвого циклу смарт-контракту

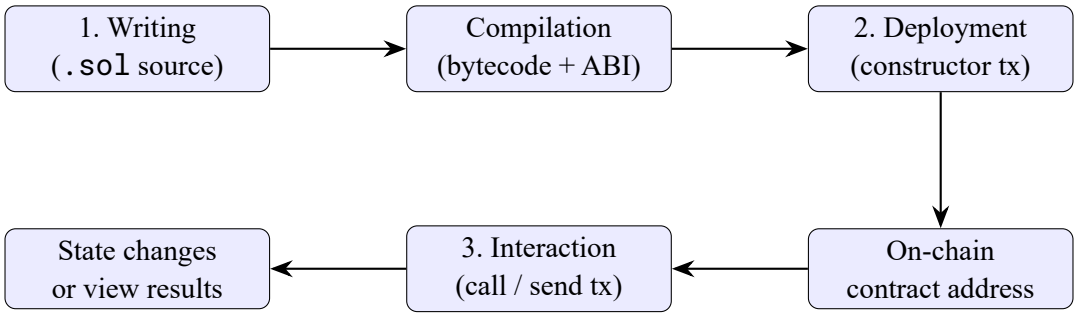

Рис. 1: Smart contract lifecycle on Ethereum.

3 Ballot Contract

Даний розділ я хочу присвятити детальному огляну оригінального контракту `Ballot` з офіційної документації Solidity 0.5.3, включаючи його структури даних, змінні стану та функції.

3.1 Призначення договору

Контракт Ballot реалізує *delegated voting* (делеговане голосування). Головуючий створює бюлетень із переліком пропозицій. Він надає право голосу окремим адресам. Кожен уповноважений виборець може або проголосувати безпосередньо за пропозицію, або делегувати (передати) свій голос іншому виборцю, якому він довіряє. В кінці будь-хто може дізнатися, яка пропозиція перемогла.

3.2 Структури даних

3.2.1 `struct Proposal` (пропозиція)

```
    struct Proposal {
        bytes32 name;      // Short name (up to 32 bytes)
        uint voteCount;    // Accumulated votes
    }
```

Тут використовується тип `bytes32` замість `string`, оскільки він займає рівно один 256-бітний слот пам'яті, тоді як `string` вимагає динамічного розподілу пам'яті і є значно дорожчим у газовому вираженні.

3.2.2 `struct Voter`

```
    struct Voter {
        uint weight;       // Voting power (0 = no right, 1+ = can vote)
        bool voted;        // True after voting or delegating
        address delegate; // Delegation target; address(0) if none
        uint vote;         // Index of chosen proposal (meaningful only
                           //   if voted==true && delegate==address(0))
    }
```

Поле	Опис
`weight`	Право голосу. Ініціалізується 0 для всіх адрес. Встановлюється 1 головуючим за допомогою функції `giveRightToVote()`. Збільшується, коли інші виборці делегують право голосу цій адресі.
`voted`	Boolean flag. Встановлюється на `true` як у випадку, коли виборець голосує сам *або* коли він делегує свої повноваження. Запобігає подвійному голосуванню.
`delegate`	Адреса Ethereum, якій виборець делегував свої повноваження. Залишається 0, якщо виборець проголосував самостійно або ще не вчинив жодних дій
`vote`	Index в масиві `proposals[]`. Використовується лише тоді, коли виборець голосує безпосередньо (не делегує).

3.3 Станові змінні

```
    address public chairperson;
    mapping(address => Voter) public voters;
    Proposal[] public proposals;
```

Змінна	Тип	Призначення
`chairperson`	`address`	Зберігає адресу розробника контракту. Тільки ця адреса може надавати право голосу.
`voters`	`mapping`	Відображає кожну адресу Ethereum у структуру `Voter`.
`proposals`	`Proposal[]`	Динамічний масив всіх пропозицій. Створюється конструктором один раз.

Примітка. *Відображення не зберігає ключі, тому неможливо пройти по всіх виборцях в ланцюжку.*

3.4 Конструктор

```
    constructor(bytes32[] memory proposalNames) public {
        chairperson = msg.sender;
        voters[chairperson].weight = 1;

        for (uint i = 0; i < proposalNames.length; i++) {
            proposals.push(Proposal({
                name: proposalNames[i],
                voteCount: 0
            }));
        }
    }
```

Конструктор виконується *exactly once* під час розгортання контракту. Він записує `msg.sender` як головуючого, автоматично надає йому вагу голосу 1 і заповнює масив `proposals` з наданих йому імен. Вартість гасу для розгортання пропорційна кількості пропозицій (кожен `push` записує новий слот пам'яті).

3.5 Функція: `giveRightToVote(address voter)`

```
    function giveRightToVote(address voter) public {
        require(msg.sender == chairperson,
                "Only chairperson can give right to vote.");
        require(!voters[voter].voted,
                "The voter already voted.");
        require(voters[voter].weight == 0);
        voters[voter].weight = 1;
    }
```

Функція застосовує три заходи безпеки:

1. Тільки головуючий може її викликати (access control).
2. Цільова адреса must not already possess voting rights (я хз як це перекласти нормально).
3. Ціль не повинна ще мати права голосу (`weight == 0`).

У разі успіху вага виборця встановлюється на 1, що дозволяє йому брати участь у голосування. Приблизна вартість гасу складає: $\sim$47 000 (запис нового слота пам'яті).

Обмеження: Кожен виборець потребує окремої транзакції. Для N виборців головуючий сплачує $N \times (\sim 47\,000 + 21\,000_{\text{base}})$ гасу. У самій документації Solidity розробники ставлять нам питання: *"Чи можете ви придумати кращий спосіб?"* – на це питання я дам в розділі з оптимізацією.

3.6 Функція: `delegate(address to)`

Це, певно, найскладніша функція в контракті. Виглядає вона наступним чином:

```
    function delegate(address to) public {
        Voter storage sender = voters[msg.sender];
        require(!sender.voted, "You already voted.");
        require(to != msg.sender, "Self-delegation is disallowed.");

        while (voters[to].delegate != address(0)) {
            to = voters[to].delegate;
            require(to != msg.sender, "Found loop in delegation.");
        }

        sender.voted = true;
        sender.delegate = to;
        Voter storage delegate_ = voters[to];

        if (delegate_.voted) {
            proposals[delegate_.vote].voteCount += sender.weight;
        } else {
            delegate_.weight += sender.weight;
        }
    }
```

1. Завантажує структуру `Voter` виборця за посиланням (`storage` pointer).
2. Перевіряє, що виборець ще не голосував і не делегує сам собі.
3. **Розв'язує ланцюжок делегування:** Цикл `while` проходить ланцюг делегування (A→B→C) до тих пір, поки не знайде виборця, який не делегує далі. Цей цикл також перевіряє наявність циклічного делегування.
4. Позначає виборця як проголосовавшого та записує остаточного делегата.
5. **Два результати:**
   - Якщо остаточний делегат вже голосував, вага виборця додається безпосередньо до обраної ним пропозиції.
   - Якщо остаточний делегат ще не голосував, вага виборця переноситься до делегата (накопичуючи voting power).

Гасові ризики: Цикл `while` зчитує storage, сплачуючи (`SLOAD` = 2 100 gas) на кожній ітерації. Ланцюжок делегування довжиною k коштує приблизно $k \times 2\,100$ gas лише за один цикл. Чим довше ланцюжок, тим більша ймовірність вичерпати гасовий ліміт блоку.

3.7 Функція: `vote(uint proposal)`

```
    function vote(uint proposal) public {
        Voter storage sender = voters[msg.sender];
        require(sender.weight != 0, "Has no right to vote");
        require(!sender.voted, "Already voted.");

        sender.voted = true;
        sender.vote = proposal;
```

```
        proposals[proposal].voteCount += sender.weight;
    }
```

Все доволі просто: позначити виборця як проголосовавшого, записати його вибір і додати його вагу до підрахунку обраної пропозиції. Якщо індекс `proposal` виходить за межі масиву, Solidity автоматично викликає `revert`, скасовуючи всі зміни стану. Вартість по гасу: ∼47 000–65 000 залежно від того, чи слоти пам'яті є новими або вже записаними.

3.8 Функції перегляду: `winningProposal()` and `winnerName()`

```
    function winningProposal() public view
        returns (uint winningProposal_)
    {
        uint winningVoteCount = 0;
        for (uint p = 0; p < proposals.length; p++) {
            if (proposals[p].voteCount > winningVoteCount) {
                winningVoteCount = proposals[p].voteCount;
                winningProposal_ = p;
            }
        }
    }

    function winnerName() public view
        returns (bytes32 winnerName_)
    {
        winnerName_ = proposals[winningProposal()].name;
    }
```

Обидві функції є типу `view`. Це означає, що вони не змінюють стан і *free to call* (без транзакції, без витрати гасу) ззовні. Функція `winningProposal()` ітерує по всіх пропозиціях та повертає індекс тієї, яка має найбільшу кількість голосів (`voteCount`). Функція `winnerName()` викликає вищезгадану для отримання індексу переможця, а потім просто повертає його ім'я.

Обмеження: Не враховує рівність голосів! Якщо є випадок, де дві пропозиції мають однакову кількість голосів, перемогу отримує та, яка має нижчий індекс.

3.9 Діаграма взаємодії контрактів

Візуалізувати це можна наступним чином:

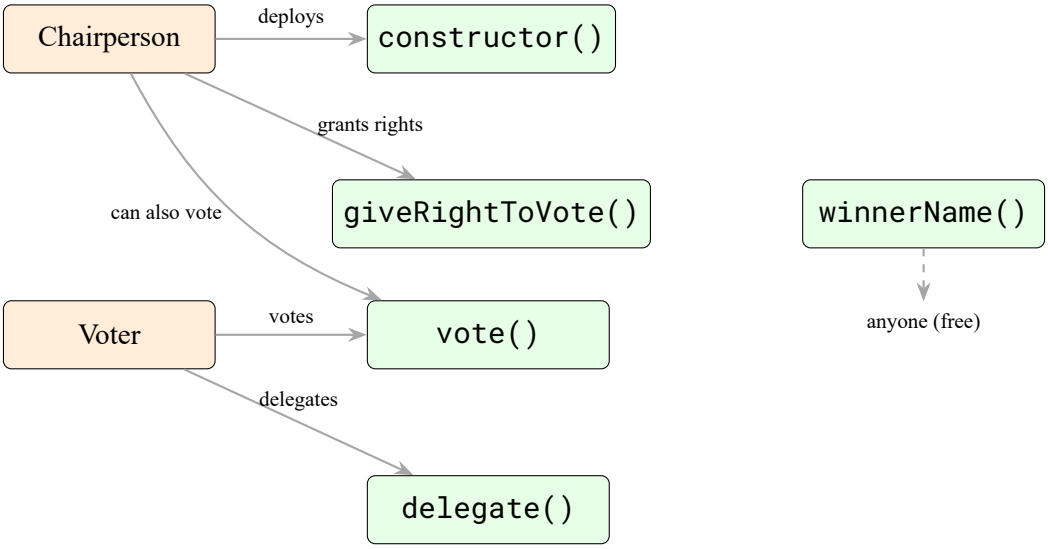

Рис. 2: Interaction roles in the Ballot contract.