Fundamentos y Aplicaciones de Blockchains

Homework 3

Depto. de Computación, UBA, 2do. Cuatrimestre 2025 *9/10/25*

Student: Juan DElia *Due: 21/10/25, 15:00 hs*

Instructions

- Upload your solution to Campus; make sure it's only one file, and clearly write your name on the first page. Name the file '<your last name>_HW3.pdf.'

  If you are proficient with LaTeX, you may also typeset your submission and submit in PDF format. To do so, uncomment the "%\begin{solution}" and "%\end{solution}" lines and write your solution between those two command lines.

- Your solutions will be graded on *correctness* and *clarity*. You should only submit work that you believe to be correct.

- You may collaborate with others on this problem set. However, you must **write up your own solutions** and **list your collaborators and any external sources (including ChatGPT and similar generative AI chatbots)** for each problem. Be ready to explain your solutions orally to a member of the course staff if asked.

This homework contains 4 questions, for a total of 65 points.

1. **Bitcoin transactions:**

   (a) (7 points) Describe the mechanism used in the actual Bitcoin application that the miners follow in order to insert transactions into a block. Would the Consistency (aka Persistence) and Liveness properties we saw in class, as well as the V, I, R functions have to be modified to capture the actual mechanism? Elaborate.

Solution: Para describir el mecanismo que usan los mineros de Bitcoin para agregar un bloque use los siguientes recursos:

En la seccion 5 del paper

El sitio de developers de Bitcoin

El mecanismo se resume en:

1. Nuevas transacciones son transmitidas en la red a los nodos.
2. Cada nodo recolecta transacciones para formar un bloque, sumando la public key a donde se debe enviar el coinbase.
3. Cada nodo intenta resolver el PoW para el target correspondiente
4. Cuando un nodo encuentra una PoW transmite el bloque en la red para todos los nodos (la primera tx es la coinbase)
5. Los nodos de la red aceptan el bloque si todas las transacciones son validas usando las merkle root y verificando que no haya double spending
6. Los nodos expresan su aceptacion de un bloque poniendose a crear el siguiente bloque de la cadena, usando como ”hash del bloque anterior” el hash del bloque que aceptan

No es necesario cambiar Consistency (If an honest miner reports a transaction tx in a block that is “deep enough” in the ledger, then tx will be reported in the same position by any honest miner in the same position in the ledger, from this round round on) ni Liveness (If all honest miners attempt to insert a transaction in the ledger, then, after u rounds, all honest miners will report it as stable). Ambas tienen sentido en el contexto del mecanismo real de bitcoin. Por como funciona el mecanismo la idea es que mediante PoW se mantenga la persistencia de las transacciones, ya que se extienden las cadenas donde se usa ”esfuerzo” si la mayoria de los nodos son honestos todos van a ver la misma cadena, o al menos el mismo prefijo (los ultimos bloques son mas volatiles).

Tampoco hace falta cambiar Liveness. Cualqueira puede enviar transacciones en la red y van a ser eventualmente incluidas.

No nos interesa cambiar la idea de las funciones, sirven para expresar la esencia del mecanismo:

- V: $V(\langle x_1, ..., x_m \rangle)$ is true if and only if the vector $V(\langle x_1, ..., x_m \rangle)$ is a valid ledeger. $V(\langle x_1, ..., x_m \rangle) \in L$

  Esto nos intteresa hacerlo. Los nodos de la red cuando verifican que todas la transacciones del ledger sean validas con las merklee root, verificando que no haya double spending, verificando que las firmas sean validas y que quienes hayan hecho transacciones tengan los fondos suficientes.

- R: If $V(\langle x_1, ..., x_m \rangle) =$ True, the value $R(C)$ is equal to $V(\langle x_1, ..., x_m \rangle)$; undefined otherwise.

  Nos interesa, en la aplicacion de blockchain cumple el proposito de listar las transacciones de los bloques.

- I: $I(st, C, round, INPUT())$ operates as follows: if the input tape contains (Insert, v), it parses v as a sequence of transactions and retains the largest subsequence $x' \leq v$ that is valid with respect to x_C (and whose transactions are not already included in x_C). Finally, $x = tx_0x'$ where tx_0 is a neutral random nonce transaction. The state st remains always ϵ.

  En este paso la definicion tambien es parecida y nos importa esta funcionalidad. Describe que selecciona un conjunto de transacciones, habria que agregar que agrega la coinbase.

(b) (3 points) Describe the purpose of a *coinbase* transaction.

Solution:

La transaccion coinbase es la primera de cada bloque y representa la recompensa que se le va a pagar a quien mine el bloque. Esta transaccion crea bitcoins, sirve no solo como recompensa sino como forma de poner moneda en circulacion, ya que no hay una autoridad central que lo haga.

2. **Proofs of work:**

(a) (5 points) Describe the purpose and implementation of the **2x1 PoW** technique.

Solution: La idea de implementar 2x1 PoW es lograr consenso soportando $\frac{1}{2}$ de nodos deshonestos, a diferencia de las soluciones que vimos que toleraban $\frac{1}{3}$.

Conceptualmente el problema es que el "backbone protocol" no asegura sufiente chain-quality. No sabemos si suficientes bloques provienen de nodos honestos.

Para garantizar que la cantidad de transacciones sea equivalente al "hashing power" de cada uno se puede implementar PoW para "minar" transacciones. Va a haber un doble proceso de mineria, uno para generar bloques y otro para insertar transacciones. Asi soporta hasta un poder de hashing menor a $\frac{1}{2}$ de los participantes maliciosos.

En lineas generales se divide en dos etapas:

- Operation: En cada ronda las "parties" corren dos protocolos en paralelo. El primero es Π_{PL} que mantiene el ledger de transacciones y requiere q queries al oraculo $H_0(.)$. El segundo proceso es el de "transaction production Π_{tx}" que continuamente genera transacciones que satisfecen $(H_1(ctr, G(nonce, v)) < T) \wedge (ctr \leq q)$ El protocolo hace q queries a $H_1(.)$
- Termination: Despues de la ronda L, una partie recolecta todas las transacciones unicas que hay en los primeros $O(k)$ bloques y retorna el valor que mas aparece de entre las transacciones

El problema de tener dos subprocesos es que un adversario podria mover su poder de computo de uno al otro.

Este protocolo que usa dos subprocesos de PoW se puede juntar en un unico proceso que usa un unico oraculo $H(.)$ con un total de q queries por ronda. Se puede conseguir cambiando la estructura de los PoWs de pares de la pinta $\langle w, ctr \rangle$ a $\langle w, ctr, label \rangle$

A grandes rasgos es como el algoritmo que conocemos de PoW pero en cada iteracion verifica si resolvio una PoW y/o la otra para un mismo target. Cuando verifica si H(ctr,h) es menor al Target lo hace tanto para ese hash como para el reverso (Una Pow para minar bloques y otra PoW para transacciones) .De esta manera verifica dos PoW **independientes** en un unico intento.

(El algoritmo esta en la pagina 36 de The Bitcoin Backbone Protocol: Analysis and Applications)

(b) (7 points) Let T_1 and T_2 be the target values in 2x1 PoW, and κ the size of the

hash function's output. What relation should they satisfy for the technique to work? Elaborate.

Solution: Tiene que valer que $T_1 = T_2 = T$ y $t > k/2$ tomando:

Sea U una variable aleatoria uniforme en el rango sobre los enteros $[0, 2^k)$ y T un entero tal que: $T = 2^t$. Para $t > k/2$ los eventos $U < T$ y $[U]^R < T$ son independientes y ambos tienen probabilidad $T \cdot 2^{-k}$

U y su reverso son los hashes que explique en el inciso anterior que buscamos para el mismo Target

Dem:

Se puede ver que cada evento sucede con probabilidad $T \cdot 2^{-k}$. La conjuncion de los eventos es elegir un entero U tal que $U < T$ y $U^R < T$. Como $T = 2^t$ la probabilidad condicional de $U < T$ deja los t bits menos significativos de U de manera unifomemente aleatoria y fija el resto de los $\kappa - t$ bits.

El caso para U^R es análogo. Quedan los t bits mas significativos de manera uniformemente aleatoria y los $\kappa - t$ restantes quedan fijos.

El evento $U^R < T$ tiene probabilidad $2^{t-(k-t)/2^t} = T \cdot 2^{-k}$ en el espacio condicional, asi que ambos eventos son independientes.

(c) (8 points) Design and argue correctness of an ℓx1 PoW scheme. Note that such a scheme would enable an ℓ-parallel blockchain. What properties of the blockchain or ledger application would, if any, benefit from a parallel blockchain? Elaborate.

Solution:

Con un ℓx1 PoW scheme se podrian hacer ℓ PoW's en paralelo, si queremos seguir con un esquema como el de 2x1 donde hay que hacer PoW para transacciones y bloques, esto implicaria que podriamos hacer $\ell/2$ de PoW para bloques y $\ell/2$ de PoW para transacciones.

De esta manera se beneficiaria la propiedad de chain-growth ya que por cada ronda se podrian minar mas bloques. Tambien se beneficiaria liveness ya que tambien se minarian mas transacciones por bloque y sin perder chain-quality ya que mantenemos el esquema de 2x1.

La correctitud de este esquema es igual al de 2x1, con un unico target podriamos definir eventos disjuntos tal que un atacante no pueda usar mas computo del que tiene a disposicion para hacer una PoW.

3. **Strong consensus:** We saw how the Bitcoin backbone protocol can be used to solve the *consensus* problem (aka *Byzantine agreement*). In the *strong consensus* problem, the *Validity* condition is strengthened to require that the output value be one of the honest parties' inputs—this property is called *Strong Validity*. (Note that this distinction is relevant only in the case of non-binary inputs.)

   (a) (5 points) What should be the assumption on the adversarial computational power (similar to the "Honest Majority Assumption") for Strong Validity to hold?

   Solution: Strong validity significa que si v es el output decidido entre las partes, v debe ser el valor inicial de alguna parte honesta. Sea V el conjunto de valores sobre los que pueden decidir las partes, la condicion que se debe cumplir para asegurar Strong validity es:

   $$n > |V|t$$

   Se puede pensar como que la proporcion de participantes deshonestos debe ser menor a $1/|V|$ pues:

   $$1/|V| > t/n$$

   Notar que si $|V| = 2$ basta con que haya una mayoria honesta.

   (b) (5 points) State and prove the Strong Validity lemma.

   Solution: *Strong Validity lemma*: Sea Π un protocolo con un conjunto de valores V que cumple las propiedades de bitcoin backbone: Chain quality, chain Growth, Common Prefix. Π cumple la propiedad de Strong Validity si $n > |V|t$.
   Prueba:
   Para que valga strong validity, sea v_i el valor mayoritario en la cadena, algun participante honesto debe haber elegido v_i. Veamoslo por absurdo:

   1. Asumamos que el output del protocolo v_{out} es un valor que no fue valor inicial de ningun participante honesto
   2. Todos los votos (bloques con valor v_{out}) fueron creados por el adversario. Notamos a ese numero como $B_{malicioso}$
   3. La forma minima para que el valor mayoritario sea el de los participantes maliciosos seria que los votos (valores) de los honestos esten equivalentemente distribuidos para cada valor v_j, menos el de los maliciososo. De esta manera si el valor outputeado v_{out} fue el de los

maliciosos vale: $B_{maliciosos} > B_{honestos}/(|V|-1)$ siendo $B_{honestos}$ el numero de "votos" que recibio cada valor de los honestos (es igual para todos).

4. Sea k la longitud de la cadena podemos reescribir

   $B_{honestos} = k - B_{maliciosos}$ y se puede desarrollar la desigualdad descripta antes:

   $B_{maliciosos} > B_{honestos}/(|V|-1) =$

   $B_{maliciosos} > (k - B_{maliciosos})/(|V|-1) =$

   $B_{maliciosos} \cdot (|V|-1) > (k - B_{maliciosos}) =$

   $B_{maliciosos} \cdot |V| > k$

   $B_{maliciosos}/k > 1/|V|$

5. Por la propiedad de chain quality con $\lambda = |V|$ por la suposicion de $n > |V|t = 1/|V| > t/n$ (es decir la proporcion de maliciosos es menor a $1/|V|$) la fraccion adversarial de bloques es menor a $\mu = 1/\lambda = 1/|V|$:

   $$B_{maliciosos}/k < 1/|V|$$

   Que es absurdo, ya que se contradice con el paso 4.

4. **Smart contract programming:** In this assignment you will create your **own custom token**. Your contract should implement the public API described below:

   - **owner:** a public payable address that defines the contract's "owner," that is, the user that deploys the contract
   - ***Transfer(address indexed from, address indexed to, uint256 value):*** an event that contains two addresses and a uint256
   - ***Mint(address indexed to, uint256 value):*** an event that contains an address and a uint256
   - ***Sell(address indexed from, uint256 value):*** an event that contains an address and a uint256
   - **totalSupply():** a view function that returns a uint256 of the total amount of minted tokens
   - **balanceOf(address_account):** a view function returns a uint256 of the amount of tokens an address owns
   - **getName():** a view function that returns a string with the token's name
   - **getSymbol():** a view function that returns a string with the token's symbol
   - **getPrice():** a view function that returns a uint128 with the token's price (at which users can redeem their tokens)
   - **transfer(address to, uint256 value):** a function that transfers *value* amount of tokens between the caller's address and the address to; if the transfer completes successfully, the function emits an event *Transfer* with the sender's and receiver's addresses and the amount of transferred tokens and returns a boolean value (*true*)
   - **mint(address to, uint256 value):** a function that enables *only the owner* to create value new tokens and give them to address to; if the operation completes successfully, the function emits an event *Mint* with the receiver's address and the amount of minted tokens and returns a boolean value (*true*)
   - **sell(uint256 value):** a function that enables a user to sell tokens for wei at a price of *600 wei per token*; if the operation completes successfully, the sold tokens are removed from the circulating supply, and the function emits an event *Sell* with the seller's address and the amount of sold tokens and returns a boolean value (*true*)
   - **close():** a function that enables *only the owner* to destroy the contract; the contract's balance in wei, at the moment of destruction, should be transferred to the owner's address
   - **fallback** functions that enable anyone to send Ether to the contract's account
   - **constructor** function that initializes the contract as needed

You should implement the smart contract and deploy it on the course's Sepolia Testnet. Your contract should be as secure and gas efficient as possible. After deploying your contract, you should buy, transfer, and sell a token in the contract.

Your contract should implement the above API **exactly as specified**. *Do not* omit implementing one of the above variables/functions/events, do not change their name or parameters, and do not add other public variables/functions. You can define other private/internal functions/variables, if necessary.

You should provide:

(a) (5 points) A detailed description of your high-level design decisions, including (but not limited to):

- What internal variables did you use?
- What is the process of buying/selling tokens and changing the price?
- How can users access their token balance?

Solution: Mis variables internas son las siguientes:

- owner: Esta varible de tipo address payable sirve para guardar la address de quien es propietario del contrato. Es necesario tener este registro ya que es el unico que puede mintear tokens y destruir el contrato.
- supply: Un uint256 que guarda cuantos tokens hay actualmente en circulacion. Es necesario para totalSupply() y es un uint256 porque es el tipo de dato que retorna la funcion.
- name y symbol: son strings necesarios para guardar el nombre y simbolo del contrato para usar en getName() y getSymbol() respectivamente.
- price: Un uint128 que es necesario para la funcion getPrice(). Se usa en sell por ejemplo para calcular cuanto eth enviarle a un usuario que vende sus tokens. Su precio es fijo.
- balances: Es un mapping de address a uint256 para llevar el registro de cuantos tokens tiene cada usuario.

No hay una manera directa de comprar tokens, la API dice que solo el owner puede mintear tokens y enviarlos a los usuarios y no hay ninguna funcion de compra implementada. Podria haber un proceso de compra paralelo entre los usuarios pero de eso no se encarga el contrato. La venta si esta en la API, un usuario puede ejecutar sell y vender sus tokens a 600 wei cada uno (siempre que el contrato tenga fondos!).

El precio del token esta fijo en el contrato, si un usuario quiere venderlo va a poder hacerlo por 600 wei.
Los usuarios pueden acceder a su balance usando la funcion balanceOf(address account) que retorna el balance de la address ingresada.
El modo de uso seria algo asi:

- Se crea el contrato con un owner (el constructor es payable asi que lo puede inicializar con eth).
- Los datos del contrato son conocidos: nombre, simbolo, suministro.
- Solo el owner puede crear tokens, otorgandoselos a todo usuario que el quiera.
- Los usuarios pueden transferirse esos tokens todo lo que quieran entre ellos.
- Los usuarios pueden vender sus tokens en el contrato a cambio de eth siempre que el contrato tenga los fondos suficientes.
- El contrato puede recibir eth mediante fallback que es payable.
- Solo owner puede destruir el contrato, recibiendo todos los eth que se encuentren dentro.

Nota: En el codigo hay un require comentado en la funcion mint que obliga al contrato a tener eth suficientes para respaldar esos tokens para intercambiarse por el eth equivalente. Esa seria una forma de hacer que todo usuario que tenga token sepa que lo va poder vender por su equivalente en eth. Como la consigna dice "implement the above API exactly as specified" y no dice nada al respecto lo deje comentado. No tener esta precaucion no necesariamente esta mal, funciona como un banco que quiza no tiene toda la liquidez necesaria para otorgar a los clientes. Pero si quisieramos el comportamiento mencionado bastaria con descomentar el require en la funcion de mint. (Puede pasar que aunque el contrato tenga mucho eth de repente muchos usuarios quieran retirar sus tokens y no alcancen los eth para todos)

(b) (5 points) A detailed gas evaluation of your implementation, including:

- The cost of deploying and interacting with your contract.
- Techniques to make your contract more cost effective.

Solution: Desplegar el contrato tuvo un costo de 1,051,307 gas (tx).

Las funciones que simplemente retornan valores del contrato no consumen gas, pues simplemente es leer el estado. Son totalSupply(), balanceOf(), getName(), getSymbol(), getPrice().

Transfer (transaccion):

El transfer, tuvo un costo de 52,356 gas.

Mint (transaccion):

Tuvo un costo de 70,900 gas.

Sell (transaccion):

Tuvo un costo de 43,607 gas.

Close (transaccion):

Tuvo un costo de 28,598 gas. (Solo hace selfdestruct)

Sell fue el mas barato porque simplemente escribe valores ya existentes. Mint fue un poco mas caro que Transfer porque hizo la primera escritura del mapping. Transfer y Mint en general consumen casi el mismo gas porque ambos actualizan dos valores. Pueden resultar mas caras las transacciones donde se agregue un nuevo usuario al mapping (como en este caso en Transfer y Mint por ser la primera vez que transaccionan dos usuarios).

Si bien sell fue la mas barata en la primera ejecucion de cada una, va a ser la mas cara siempre que transfer y Mint no agreguen usuarios al struct, pues las tres actualizan dos valores pero sell ademas hace transfer.

Una tecnica que use para hacer mas efectivo el costo del contrato fue que el constructor sea payable, asi el owner puede crear el contrato y fondearlo en una misma transaccion.

(c) (5 points) A thorough listing of potential hazards and vulnerabilities that can occur in the smart contract and a detailed analysis of the security mechanisms that can mitigate these hazards.

Solution: Voy a listar algunos posibles peligros que fui encontrando y como los mitigue:

- Re-entrancy: En sell use transfer que deberia ser segura en este sentido. Ademas antes de enviar los eth correspondientes reste los tokens correspondientes al usuario y antes de transferir se verifica que el balance de tokens sea mayor a la cantidad a vender, asi que aunque llegara a haber reentrancy es seguro.
- Privilegios: Necesitamos que solo el owner pueda mintear y destruir el contrato. Para que no lo pueda hacer cualquiera se usan requires en

mint() y close() que verifican que quien invoque a las funciones sea el owner.

- Como menciones en el inciso a) podria no haber suficiente eth para que todos vendan sus tokens al precio indicado si todos decidieran vender a la vez. Esto no esta mitigado pero como ya explique es tan sencillo como agregar un require en mint que verifique que el eth del contrato respalde al supply total. No lo considero una vulnerabilidad pero si es algo a tener en cuenta si se quisiera un comportamiento diferente

(d) (5 points) The transaction history of the deployment of and interaction with your contract.

Solution: Hice una ejecucion simple donde se crea el contrato, el owner mintea tokens para un usuario, ese usuario le envia de vuelta algunos tokens al owner, un usuario vende algunos tokens, el owner cierra el contrato.
Las respectivas transaccions:

1. Despligue del contrato , tx:
   0xb3e37fa5b9dfa2f19fd1f8d883833126f90ae524427002b7af2e7886667f1d96
2. Mint , tx:
   0x246703608a107bad11410a6b36a929e8523ed7765c4fb36f2d237b1fab649ace
3. Transfer , tx:
   0xb14fa084b1fa4469858a9ba1e899c42b582f734d6ff13e3bedda58e5551cb4e
4. Sell , tx:
   0x2e6aa9818d2a3463667e4c7896b7e1aba24926a591bf078cced1cb33908955f7
5. Close , tx:
   0xc018ec2a09ced8735b8fb0153189203bfb1d55ed1aed16a388753d251d9a0cac

(e) (5 points) The code of your contract.

Solution: En mi repositorio de github esta el smart contract junto a algunos tests para su funcionalidad basica. Se pueden correr con 'npx hardhat test'.
Lo dejo copiado aca abajo:

```
// SPDX-License-Identifier: UNLICENSED
```

```
pragma solidity ^0.8.28;

contract Token {
    address payable public owner;
    uint256 supply;
    string name;
    string symbol;
    uint128 price = 600 wei;

    mapping(address => uint256) private balances;

    event Transfer(address indexed from, address indexed to, uint256 value);
    event Mint(address indexed to, uint256 value);
    event Sell(address indexed from, uint256 value);

    constructor(string memory _name, string memory _symbol) payable {
        name = _name;
        symbol = _symbol;
        owner = payable(msg.sender);
    }

    fallback() external payable {}

    function totalSupply() public view returns (uint256) {
        return supply;
    }

    function balanceOf(address _account) public view returns (uint256) {
        return balances[_account];
    }

    function getName() public view returns (string memory) {
        return name;
    }

    function getSymbol() public view returns (string memory) {
        return symbol;
    }

    function getPrice() public view returns (uint128) {
        return price;
    }
```

```
function transfer(address to, uint256 value) public returns (bool) {
    require(balanceOf(msg.sender) >= value, "Balance insuficiente");
    balances[msg.sender] -= value;
    balances[to] += value;

    emit Transfer(msg.sender, to, value);
    return true;
}

function mint(address to, uint256 value) public returns (bool) {
    require(msg.sender == owner, "solo el owner puede mintear tokens");
    // Sigo la especificaion al pie de la letra, si quisiera que el contrato
    //token deberia agregar el require:
    // require(
    //     address(this).balance >= (totalSupply() + value) * getPrice(),
    //     "El contrato no tiene fondos suficientes para respaldar los tokens
    // );
    supply += value;
    balances[to] += value;

    emit Mint(to, value);
    return true;
}

function sell(uint256 value) public returns (bool) {
    require(value > 0, "Debes vender al menos un token");
    require(balanceOf(msg.sender) >= value, "Balance insuficiente");
    uint256 ethers = value * getPrice();
    //no deberia pasar nunca
    require(
        address(this).balance >= ethers,
        "el contrato tiene fondos insuficientes"
    );
    balances[msg.sender] -= value;
    supply -= value;
    payable(msg.sender).transfer(ethers);

    emit Sell(msg.sender, value);
    return true;
}

function close() public {
    require(
```

```
            msg.sender == owner,
            "solo el owner puede destruir el contrato"
        );

        selfdestruct(owner);
    }
}
```