TP BITCOIN

Integrantes:

- Fabián Colque
- Nicolas Vagó
- Sofia Marchesini
- Sebastian Makkos

Agenda

- Introducción (fabian
- Diagramas (fabian y seba
- Implementación
  - Peer discovery (nico
  - Handshake (nico
  - Block Download (nico
  - Header Validation (seba
    - proof of work (sofi
  - Block Broadcasting (nico
  - Block Validation :
    - Merkle tree (fabian y sofi
    - Proof of Inclusion (sofi
  - Wallet y sus funcionalidades(sofi
  - UTXO Set (sofi
  - Archivo de configuración (fabian
  - Archivo de log (sofi
- Utilización de hilos (Fabian
- Channels (Fabian
- GTK & Glade (Seba
- Demo del programa (
- Forma de trabajo (Seba
- Referencias (Seba
- Espacio de preguntas

Introducción

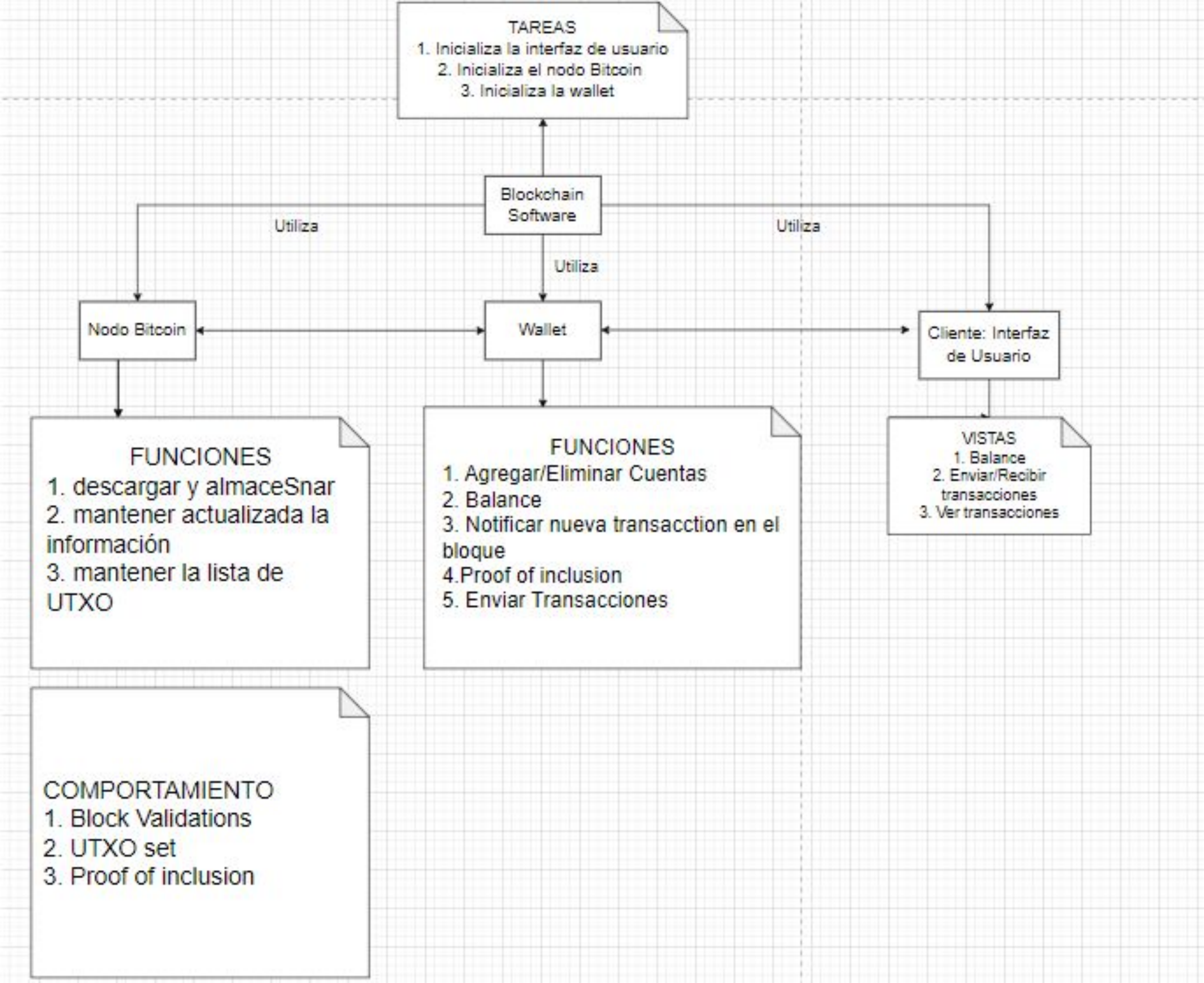
El objetivo del proyecto es la implementación de un nodo Bitcoin con funcionalidades acotadas en lenguaje Rust.

Funcionalidades

- Descargar y almacenar cadena de headers
- Actualización de bloques y transacciones
- Mantener lista de UTXO

Diagramas

Blockchain Software



TAREAS
1. Inicializa la interfaz de usuario
2. Inicializa el nodo Bitcoin
3. Inicializa la wallet

FUNCIONES
1. descargar y almaceSnar
2. mantener actualizada la información
3. mantener la lista de UTXO

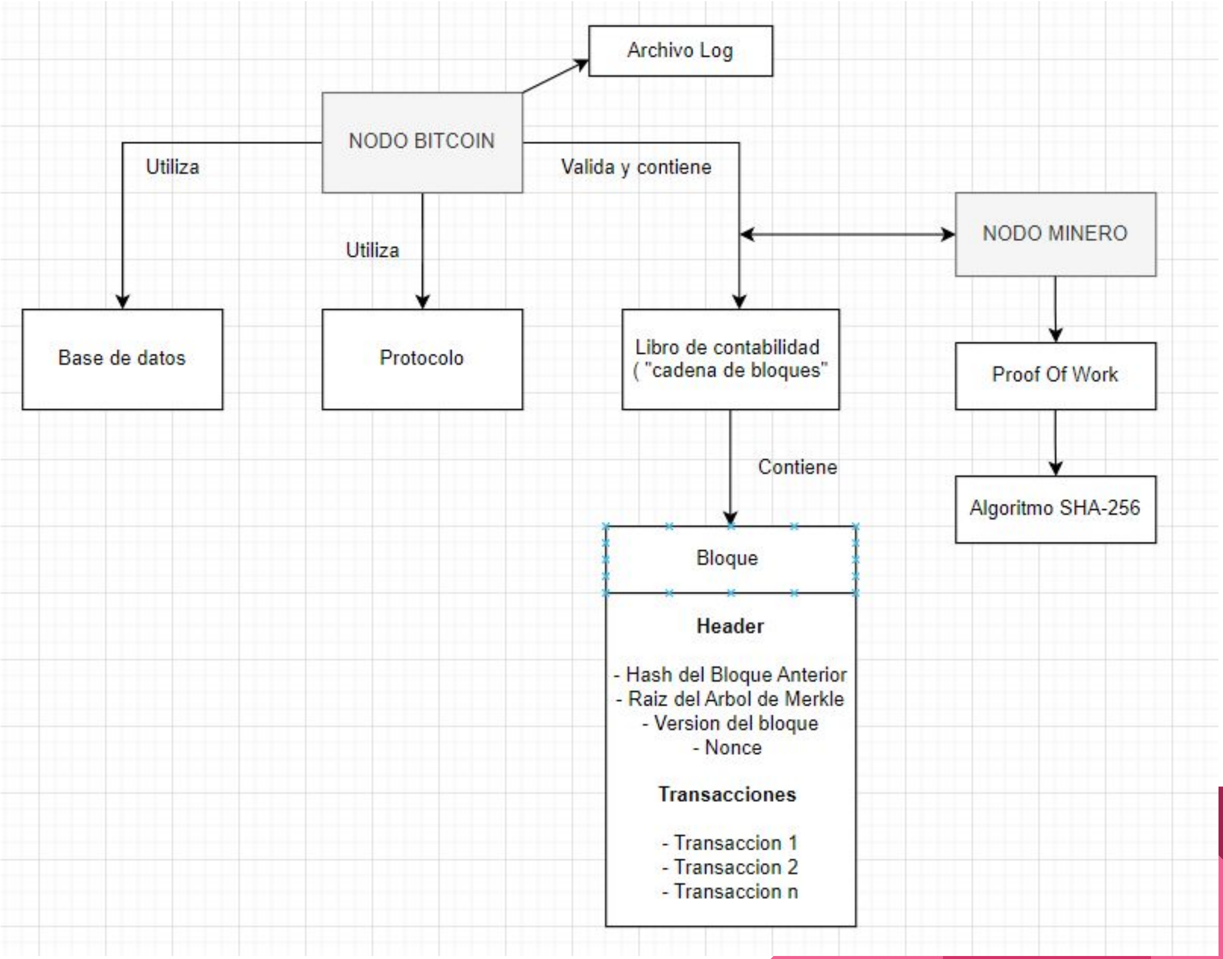
FUNCIONES
1. Agregar/Eliminar Cuentas
2. Balance
3. Notificar nueva transacction en el bloque
4.Proof of inclusion
5. Enviar Transacciones

VISTAS
1. Balance
2. Enviar/Recibir transacciones
3. Ver transacciones

COMPORTAMIENTO
1. Block Validations
2. UTXO set
3. Proof of inclusion

Diagramas

Nodo Bitcoin



Header

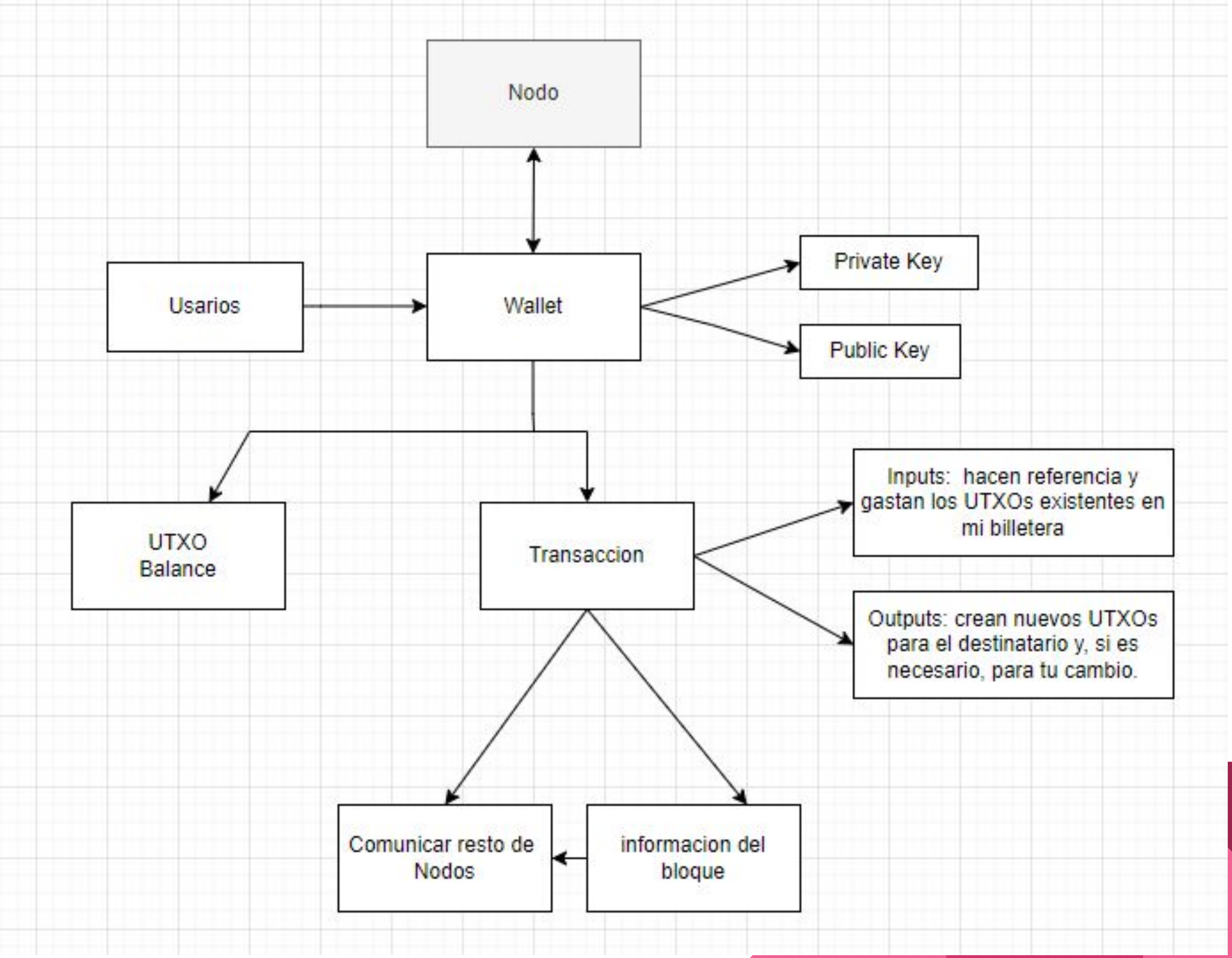
- Hash del Bloque Anterior
- Raiz del Arbol de Merkle
- Version del bloque
- Nonce

Transacciones

- Transaccion 1
- Transaccion 2
- Transaccion n

Diagramas

Wallet


Nodo
Usarios
Wallet
Private Key
Public Key
UTXO Balance
Transaccion
Inputs: hacen referencia y gastan los UTXOs existentes en mi billetera
Outputs: crean nuevos UTXOs para el destinatario y, si es necesario, para tu cambio.
Comunicar resto de Nodos
informacion del bloque

Block Download

Blocks First

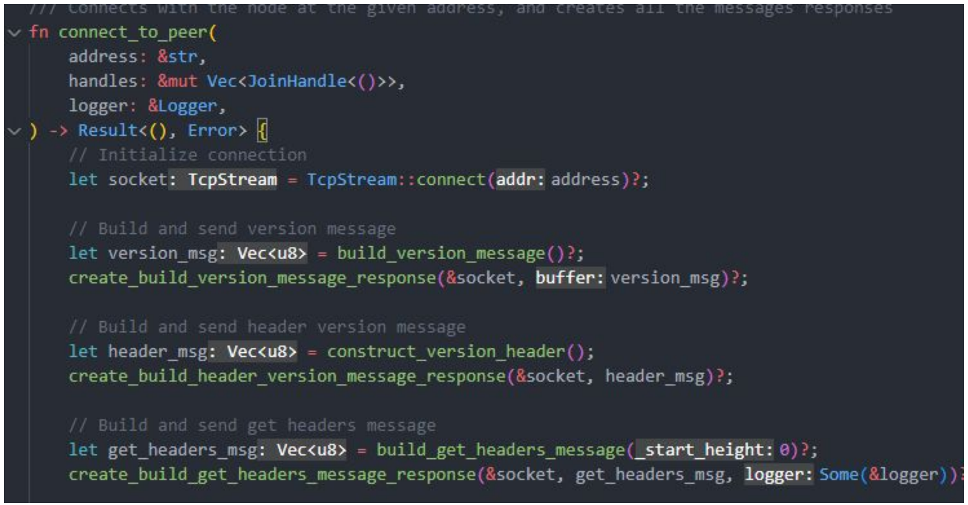
-> GetBlocks -> Inventory -> GetData -> Download Blocks

Headers First

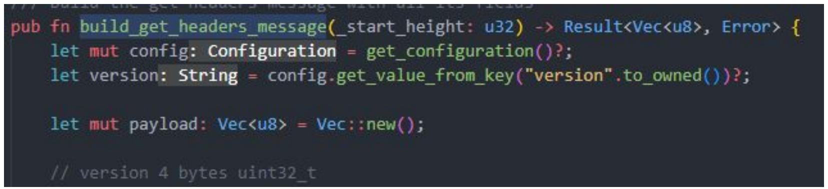
-> GetHeaders -> Validations -> Download Blocks

Block Download : Header First

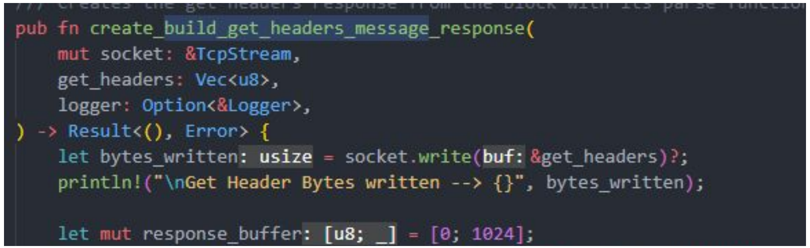
Header Download (getHeaders)



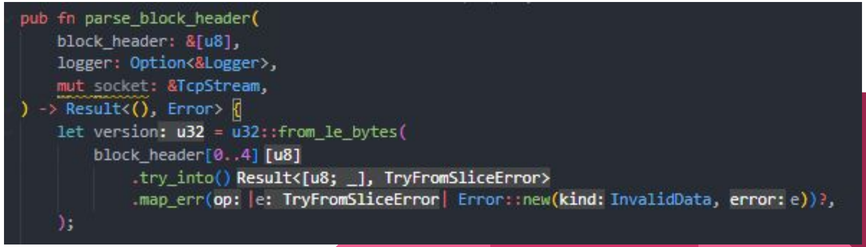
```
/// Connects with the node at the given address, and creates all the messages responses
fn connect_to_peer(
    address: &str,
    handles: &mut Vec<JoinHandle<()>>,
    logger: &Logger,
) -> Result<(), Error> {
    // Initialize connection
    let socket: TcpStream = TcpStream::connect(addr: address)?;

    // Build and send version message
    let version_msg: Vec<u8> = build_version_message()?;
    create_build_version_message_response(&socket, buffer: version_msg)?;

    // Build and send header version message
    let header_msg: Vec<u8> = construct_version_header();
    create_build_header_version_message_response(&socket, header_msg)?;

    // Build and send get headers message
    let get_headers_msg: Vec<u8> = build_get_headers_message(_start_height: 0)?;
    create_build_get_headers_message_response(&socket, get_headers_msg, logger: Some(&logger))
```



```
/// Build the get headers message with all its fields
pub fn build_get_headers_message(_start_height: u32) -> Result<Vec<u8>, Error> {
    let mut config: Configuration = get_configuration()?;
    let version: String = config.get_value_from_key("version".to_owned())?;

    let mut payload: Vec<u8> = Vec::new();

    // version 4 bytes uint32_t
```



```
/// Creates the get headers response from the block with its parse function
pub fn create_build_get_headers_message_response(
    mut socket: &TcpStream,
    get_headers: Vec<u8>,
    logger: Option<&Logger>,
) -> Result<(), Error> {
    let bytes_written: usize = socket.write(buf: &get_headers)?;
    println!("\nGet Header Bytes written --> {}", bytes_written);

    let mut response_buffer: [u8; _] = [0; 1024];
```



```
pub fn parse_block_header(
    block_header: &[u8],
    logger: Option<&Logger>,
    mut socket: &TcpStream,
) -> Result<(), Error> {
    let version: u32 = u32::from_le_bytes(
        block_header[0..4] [u8]
            .try_into() Result<[u8; _], TryFromSliceError>
            .map_err(op: |e: TryFromSliceError| Error::new(kind: InvalidData, error: e))?,
    );
```

Peer Discovery

Objetivo: Reconocer nodos semilla con los cuales interactuar .

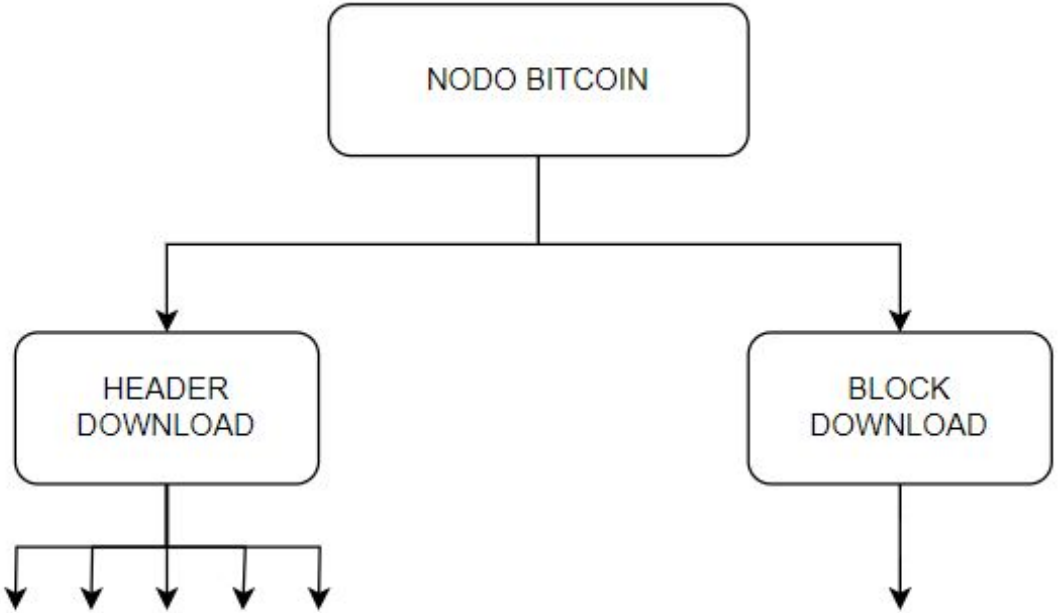
- Obtención de direcciones IP de peers a través de *dirección DNS*
- Utilización librería *STD::NET*
- Utilización *to_socket_addr*

Handshake

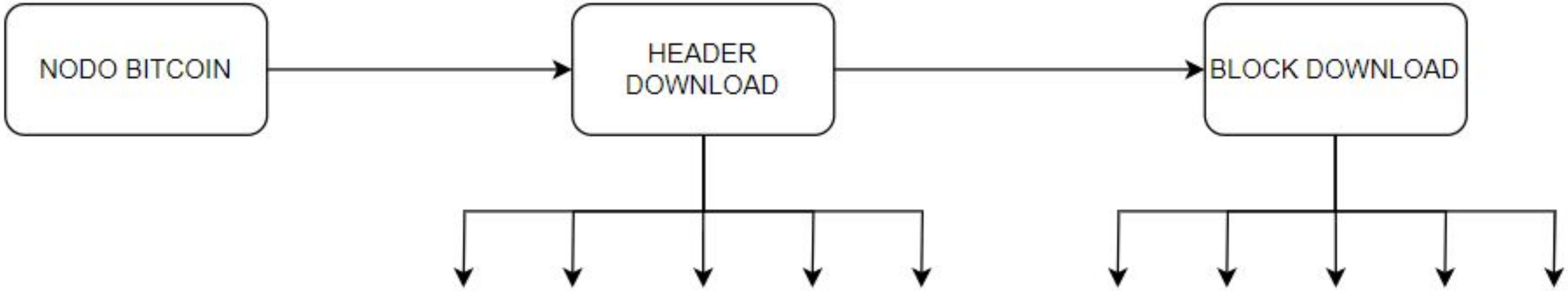
Objetivo: Crear un pool de TcpStreams listos intercambiar mensajes con peers

- Establecer *Handshake* con peers
- Seguimiento del protocolo
- Lista de Structs *TcpStream* para utilizarlos en las distintas features
- Funcion *set_tcp_vec_stream (direcciones_ip_nodos)*
- Función *Handshake (Struct TcpStream)*

Descarga Paralelizada - Primer enfoque

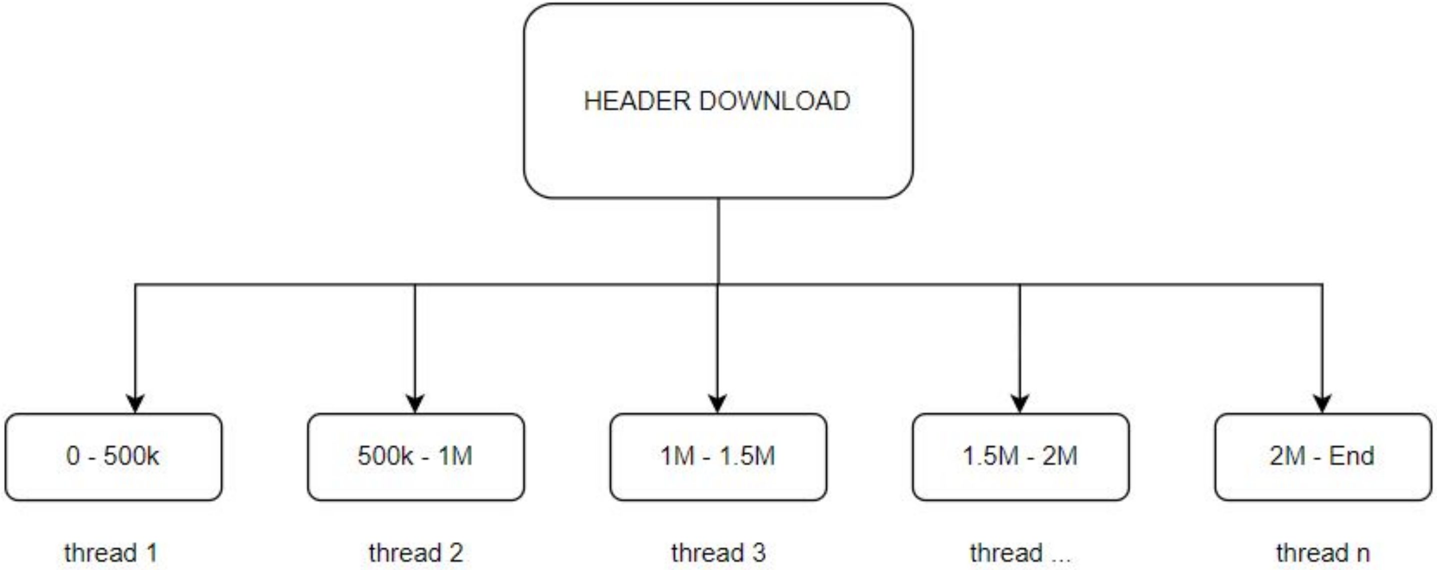


Descarga Paralelizada - Modelo final



Descarga Paralelizada

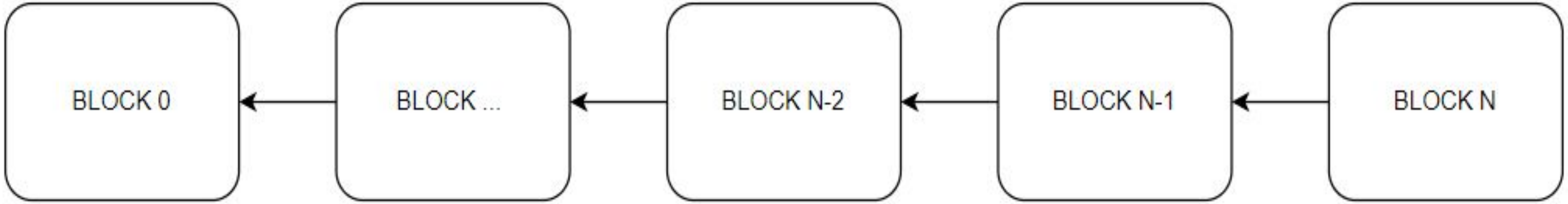
Descarga de headers [*getheaders message*] .



Descarga Paralelizada

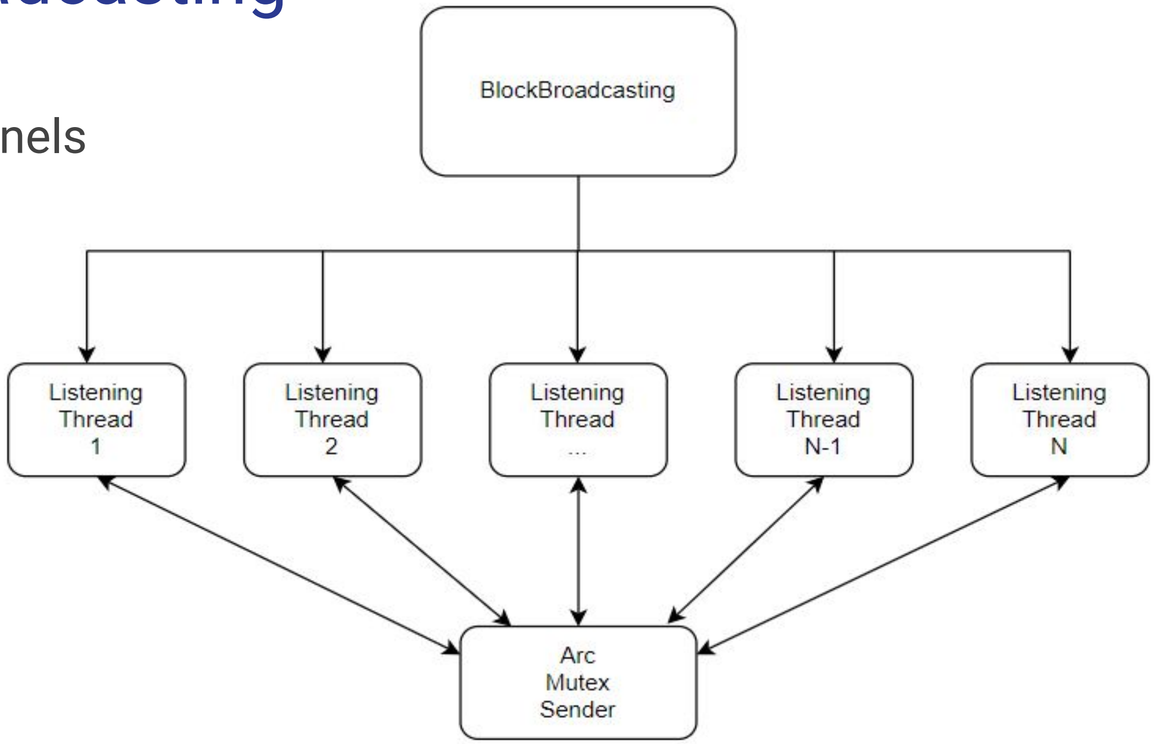
Descarga de blocks [*getdata message*] .

Primer enfoque secuencial. Pros: En paralelo a la descarga de headers. Contras: Secuencialidad y poco robusto

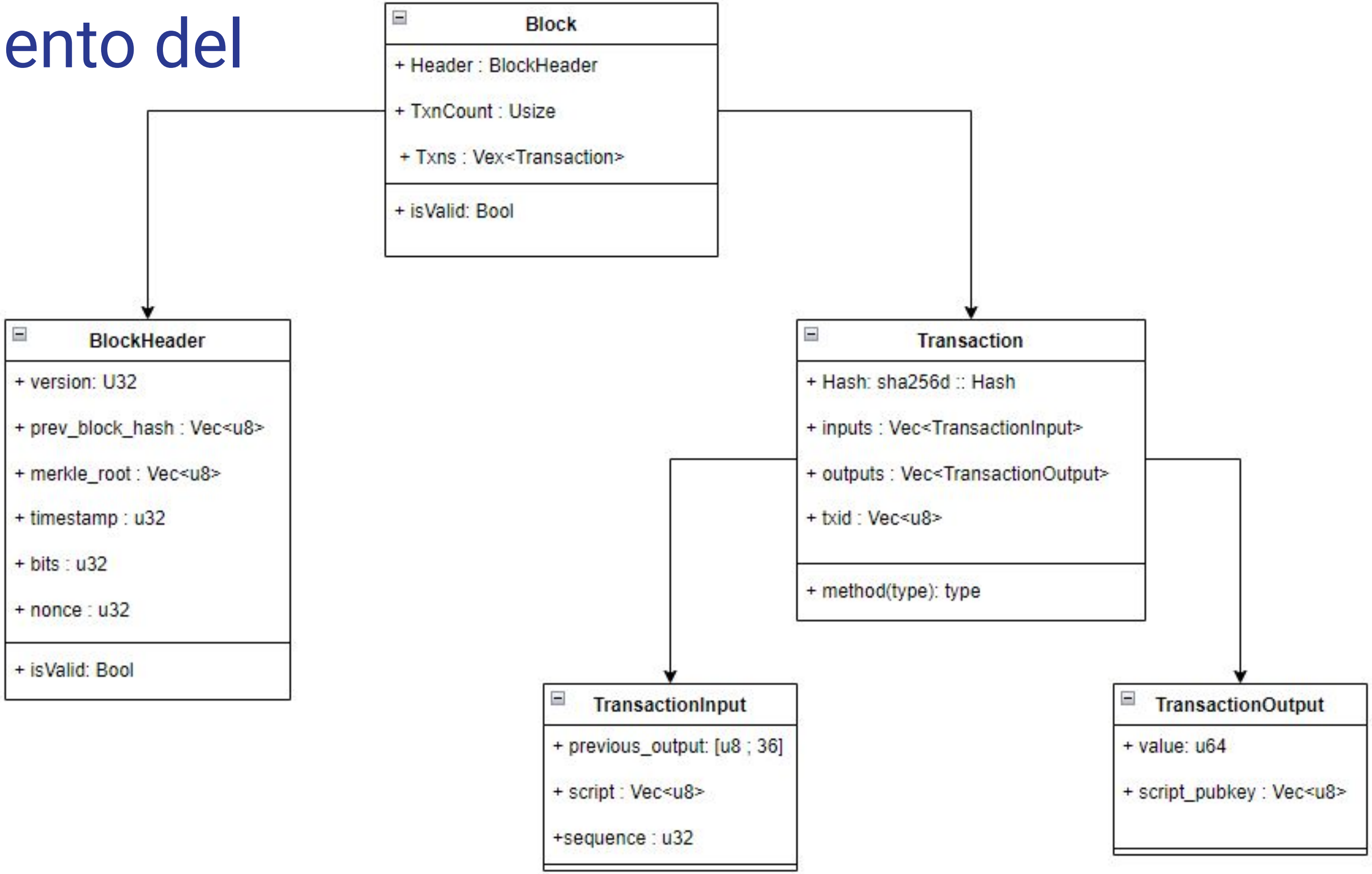


Block Broadcasting

Threads y Channels



Comportamiento del NODO



Block

+ Header : BlockHeader
+ TxnCount : Usize
+ Txns : Vex<Transaction>

+ isValid: Bool

BlockHeader

+ version: U32
+ prev_block_hash : Vec<u8>
+ merkle_root : Vec<u8>
+ timestamp : u32
+ bits : u32
+ nonce : u32

+ isValid: Bool

Transaction

+ Hash: sha256d :: Hash
+ inputs : Vec<TransactionInput>
+ outputs : Vec<TransactionOutput>
+ txid : Vec<u8>

+ method(type): type

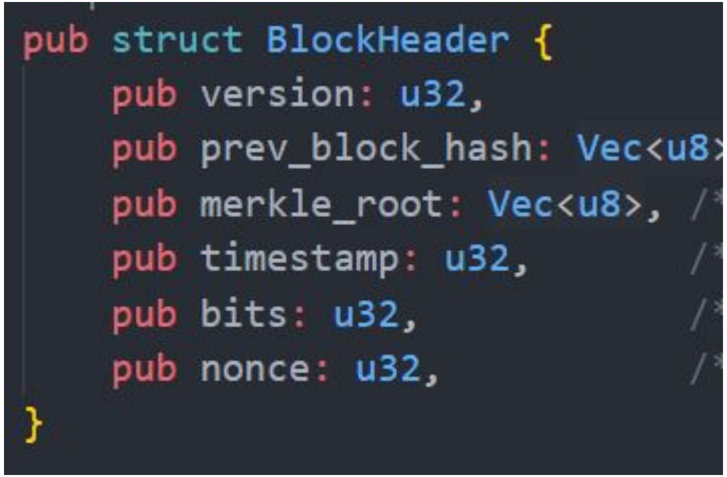
TransactionInput

+ previous_output: [u8 ; 36]
+ script : Vec<u8>
+sequence : u32

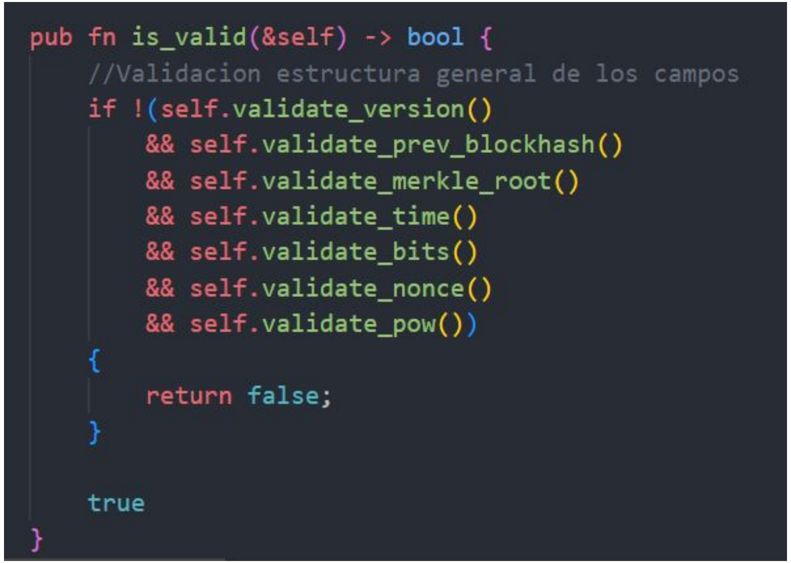
TransactionOutput

+ value: u64
+ script_pubkey : Vec<u8>

Header Validation

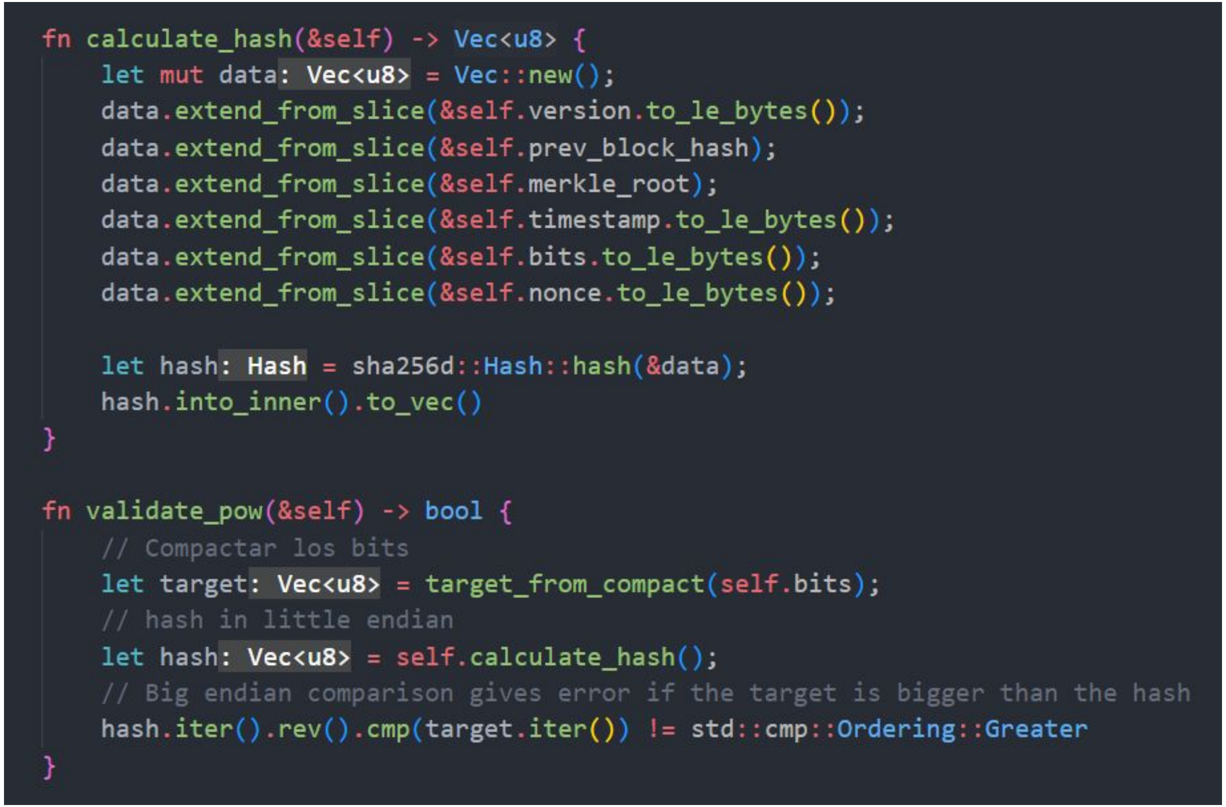


```
pub struct BlockHeader {
    pub version: u32,
    pub prev_block_hash: Vec<u8
    pub merkle_root: Vec<u8>, /*
    pub timestamp: u32,       /*
    pub bits: u32,            /*
    pub nonce: u32,           /*
}
```



```
pub fn is_valid(&self) -> bool {
    //Validacion estructura general de los campos
    if !(self.validate_version()
        && self.validate_prev_blockhash()
        && self.validate_merkle_root()
        && self.validate_time()
        && self.validate_bits()
        && self.validate_nonce()
        && self.validate_pow())
    {
        return false;
    }

    true
}
```

Proof of Work

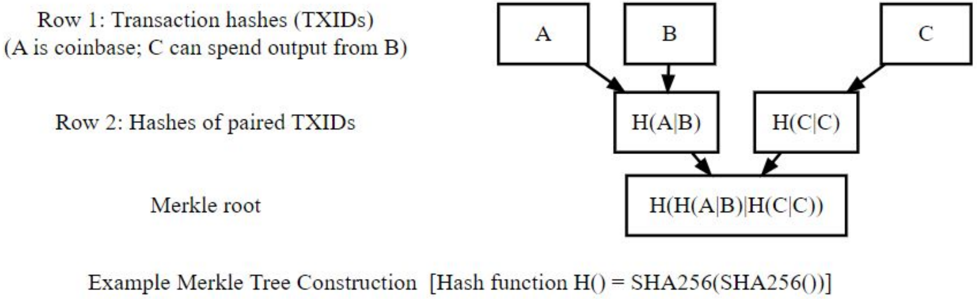


```
fn calculate_hash(&self) -> Vec<u8> {
    let mut data: Vec<u8> = Vec::new();
    data.extend_from_slice(&self.version.to_le_bytes());
    data.extend_from_slice(&self.prev_block_hash);
    data.extend_from_slice(&self.merkle_root);
    data.extend_from_slice(&self.timestamp.to_le_bytes());
    data.extend_from_slice(&self.bits.to_le_bytes());
    data.extend_from_slice(&self.nonce.to_le_bytes());

    let hash: Hash = sha256d::Hash::hash(&data);
    hash.into_inner().to_vec()
}

fn validate_pow(&self) -> bool {
    // Compactar los bits
    let target: Vec<u8> = target_from_compact(self.bits);
    // hash in little endian
    let hash: Vec<u8> = self.calculate_hash();
    // Big endian comparison gives error if the target is bigger than the hash
    hash.iter().rev().cmp(target.iter()) != std::cmp::Ordering::Greater
}
```

el hash del bloque debe ser menor o igual que el objetivo de dificultad para que la prueba de trabajo sea válida

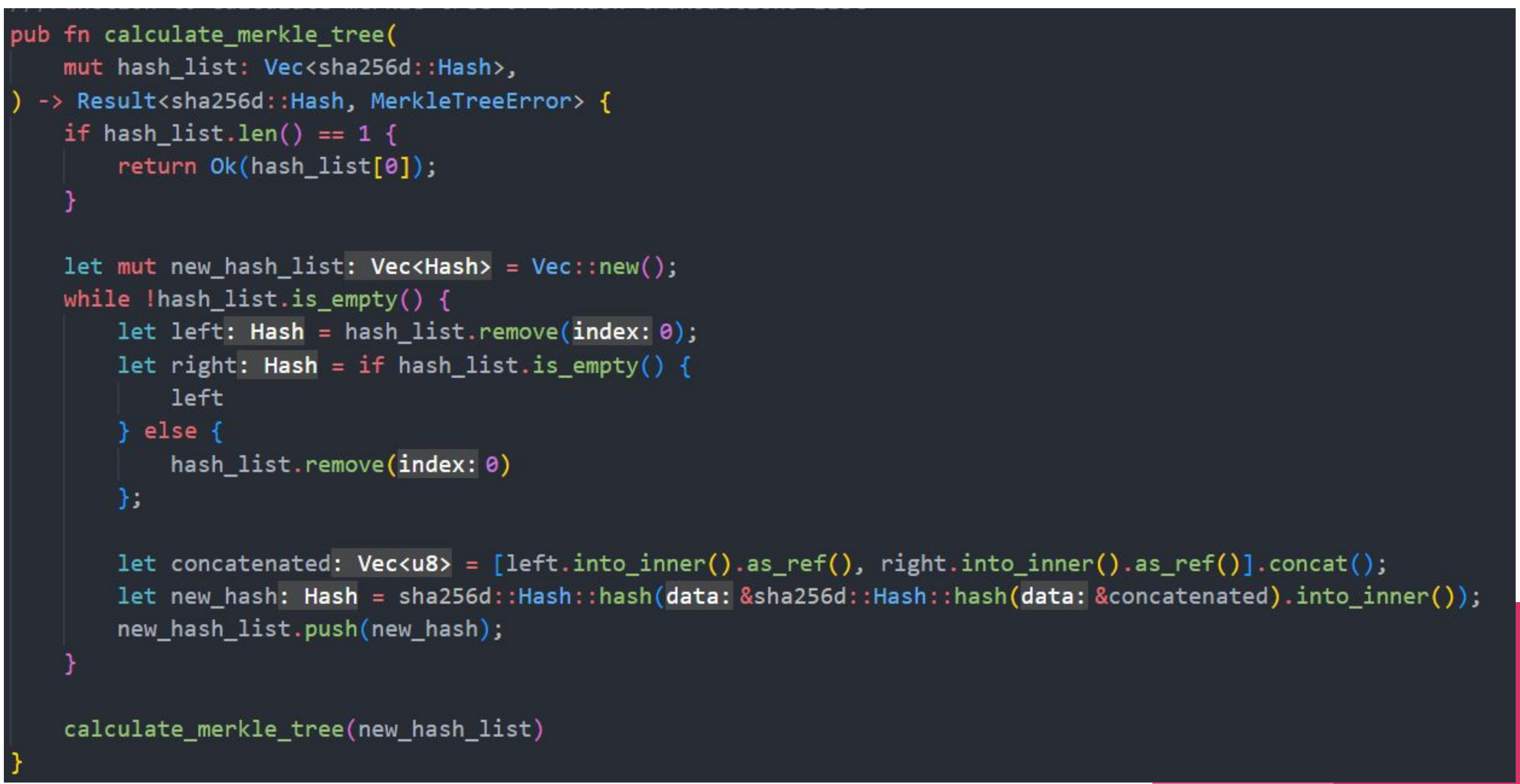
Merkle Tree

Implementación del árbol sin persistencia de los nodos intermedias en una estructura en particular


Row 1: Transaction hashes (TXIDs)
(A is coinbase; C can spend output from B)
A
B
C
Row 2: Hashes of paired TXIDs
H(A|B)
H(C|C)
Merkle root
H(H(A|B)|H(C|C))
Example Merkle Tree Construction [Hash function H() = SHA256(SHA256())]

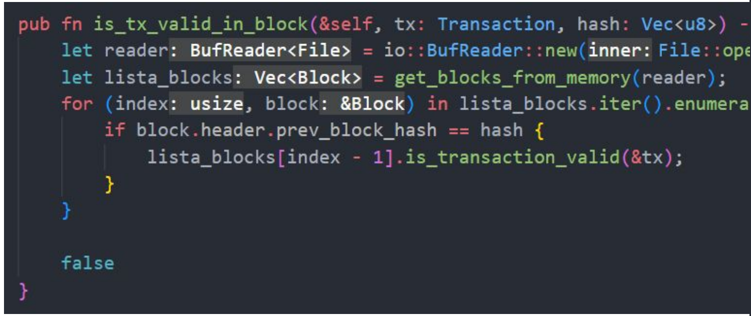
Example Merkle Tree Construction

Merkle Tree

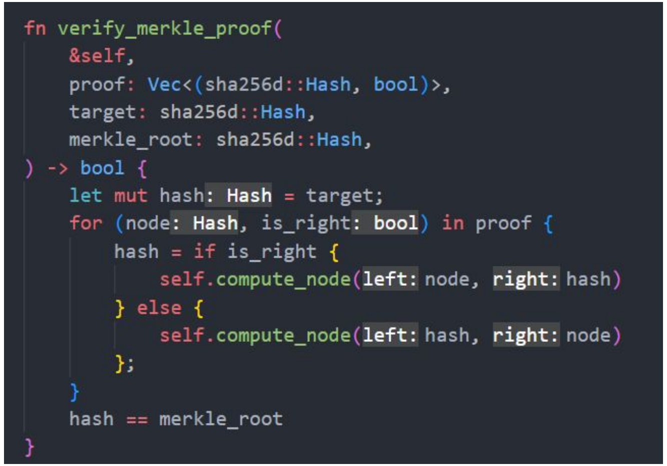


```rust
pub fn calculate_merkle_tree(
    mut hash_list: Vec<sha256d::Hash>,
) -> Result<sha256d::Hash, MerkleTreeError> {
    if hash_list.len() == 1 {
        return Ok(hash_list[0]);
    }

    let mut new_hash_list: Vec<Hash> = Vec::new();
    while !hash_list.is_empty() {
        let left: Hash = hash_list.remove(index: 0);
        let right: Hash = if hash_list.is_empty() {
            left
        } else {
            hash_list.remove(index: 0)
        };

        let concatenated: Vec<u8> = [left.into_inner().as_ref(), right.into_inner().as_ref()].concat();
        let new_hash: Hash = sha256d::Hash::hash(data: &sha256d::Hash::hash(data: &concatenated).into_inner());
        new_hash_list.push(new_hash);
    }

    calculate_merkle_tree(new_hash_list)
}
```

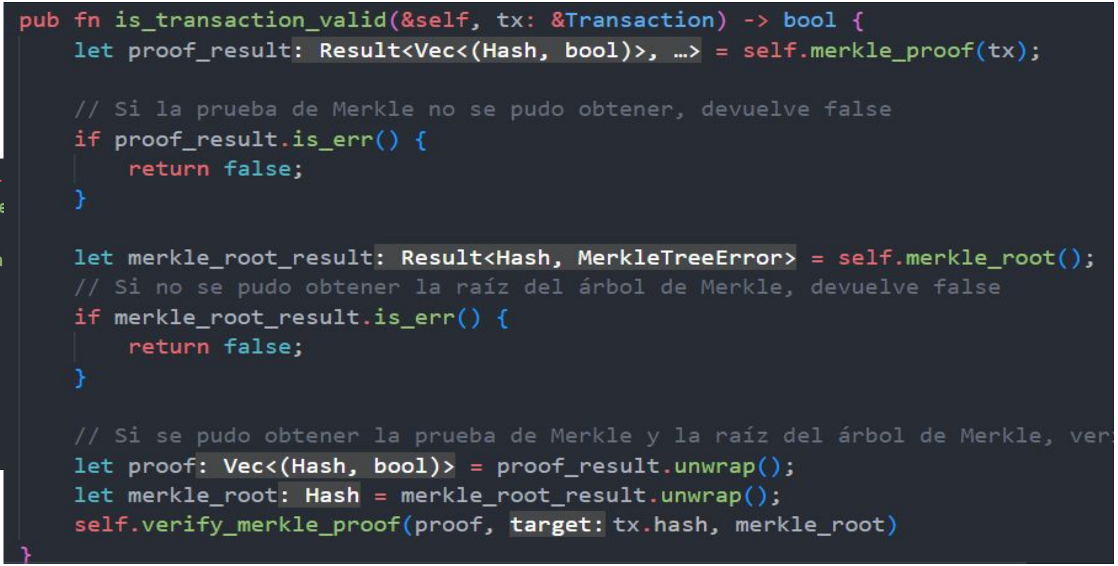
Proof Of Inclusion



```
pub fn is_tx_valid_in_block(&self, tx: Transaction, hash: Vec<u8>) -
    let reader: BufReader<File> = io::BufReader::new(inner: File::ope
    let lista_blocks: Vec<Block> = get_blocks_from_memory(reader);
    for (index: usize, block: &Block) in lista_blocks.iter().enumera
        if block.header.prev_block_hash == hash {
            lista_blocks[index - 1].is_transaction_valid(&tx);
        }
    }

    false
}
```



```
fn verify_merkle_proof(
    &self,
    proof: Vec<(sha256d::Hash, bool)>,
    target: sha256d::Hash,
    merkle_root: sha256d::Hash,
) -> bool {
    let mut hash: Hash = target;
    for (node: Hash, is_right: bool) in proof {
        hash = if is_right {
            self.compute_node(left: node, right: hash)
        } else {
            self.compute_node(left: hash, right: node)
        };
    }
    hash == merkle_root
}
```

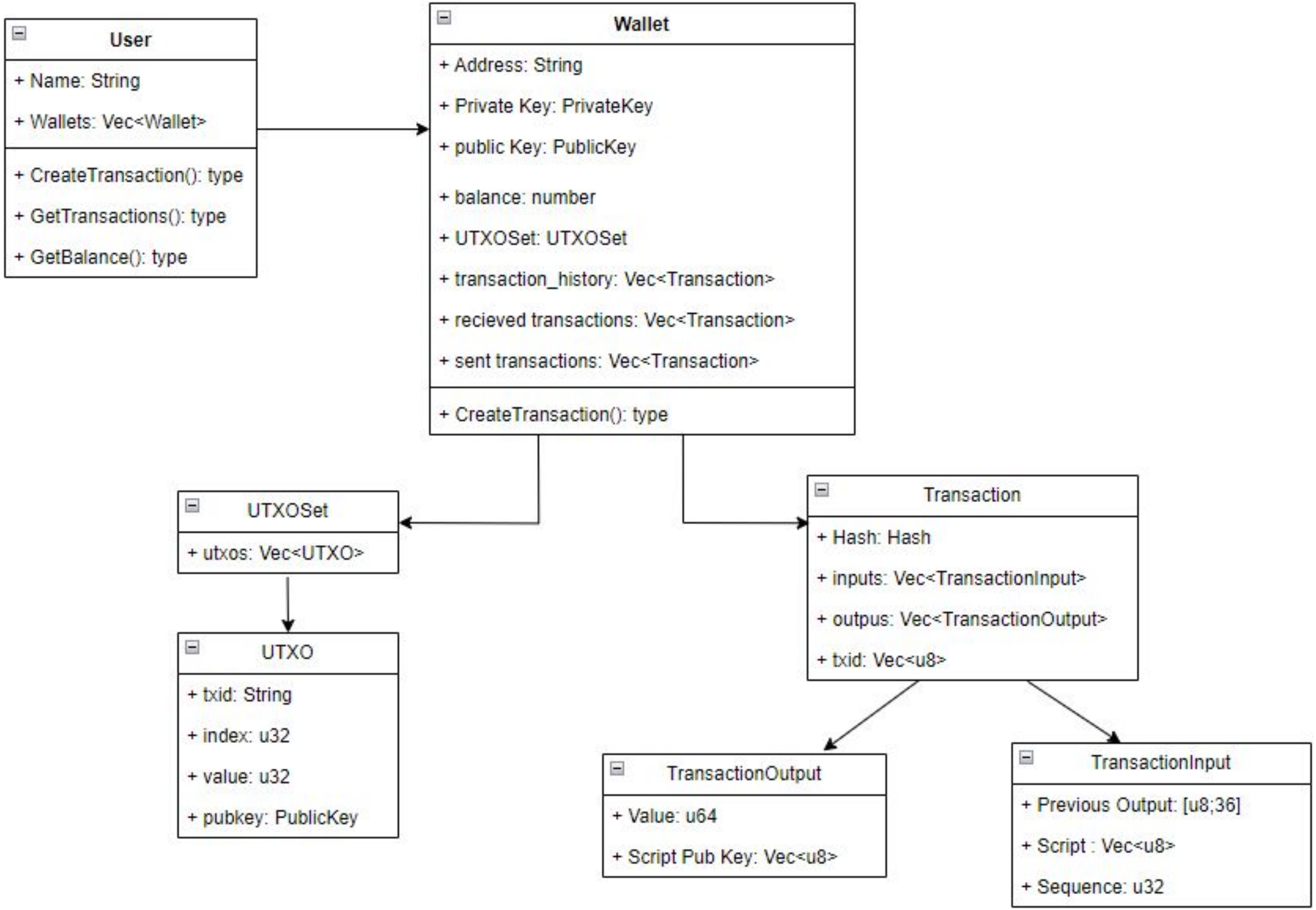


```
pub fn is_transaction_valid(&self, tx: &Transaction) -> bool {
    let proof_result: Result<Vec<(Hash, bool)>, …> = self.merkle_proof(tx);

    // Si la prueba de Merkle no se pudo obtener, devuelve false
    if proof_result.is_err() {
        return false;
    }

    let merkle_root_result: Result<Hash, MerkleTreeError> = self.merkle_root();
    // Si no se pudo obtener la raíz del árbol de Merkle, devuelve false
    if merkle_root_result.is_err() {
        return false;
    }

    // Si se pudo obtener la prueba de Merkle y la raíz del árbol de Merkle, ver
    let proof: Vec<(Hash, bool)> = proof_result.unwrap();
    let merkle_root: Hash = merkle_root_result.unwrap();
    self.verify_merkle_proof(proof, target: tx.hash, merkle_root)
}
```

Se obtiene la prueba de merkle y luego la raiz de merkle . Luego Calcula el hash a través de la prueba de Merkle y verifica si el hash resultante coincide con la raíz del árbol de Merkle

El usuario podra pedir una prueba de inclusión de una transacción en un bloque, y verificarla localmente

Wallet



User

+ Name: String
+ Wallets: Vec<Wallet>

+ CreateTransaction(): type
+ GetTransactions(): type
+ GetBalance(): type

Wallet

+ Address: String
+ Private Key: PrivateKey
+ public Key: PublicKey
+ balance: number
+ UTXOSet: UTXOSet
+ transaction_history: Vec<Transaction>
+ recieved transactions: Vec<Transaction>
+ sent transactions: Vec<Transaction>

+ CreateTransaction(): type

UTXOSet

+ utxos: Vec<UTXO>

UTXO

+ txid: String
+ index: u32
+ value: u32
+ pubkey: PublicKey

Transaction

+ Hash: Hash
+ inputs: Vec<TransactionInput>
+ outpus: Vec<TransactionOutput>
+ txid: Vec<u8>

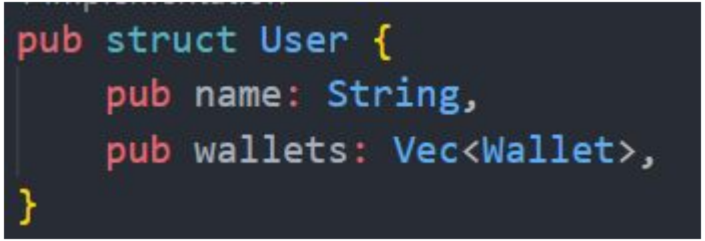
TransactionOutput

+ Value: u64
+ Script Pub Key: Vec<u8>

TransactionInput

+ Previous Output: [u8;36]
+ Script : Vec<u8>
+ Sequence: u32

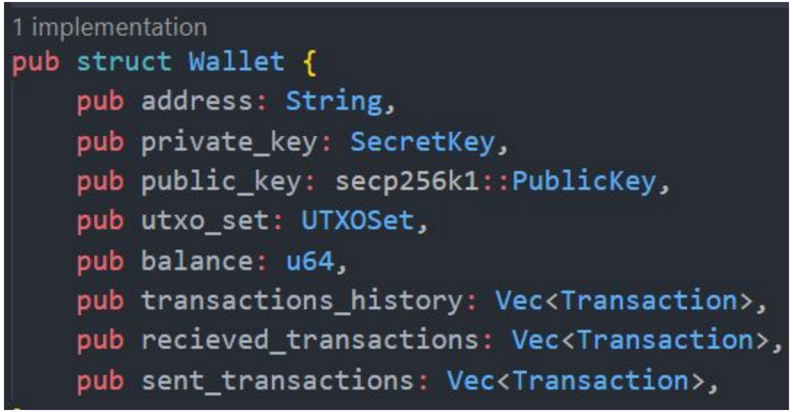
Wallet



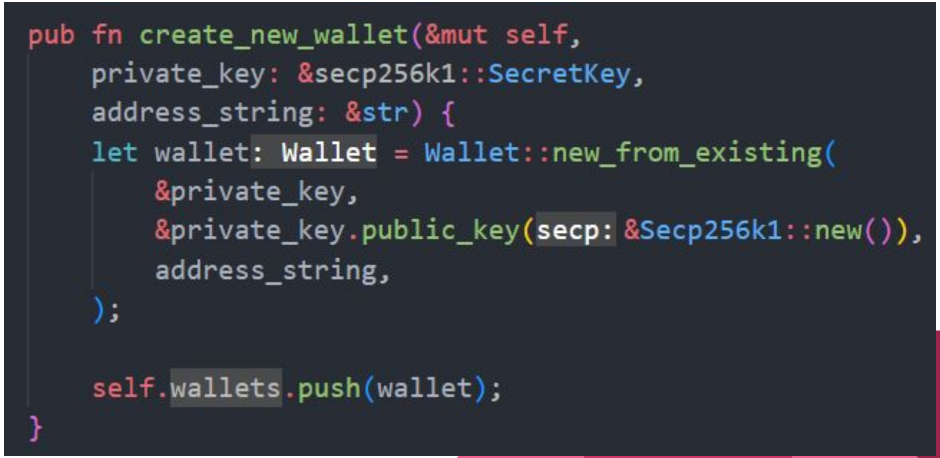
```
pub struct User {
    pub name: String,
    pub wallets: Vec<Wallet>,
}
```

El usuario podra ingresar una o mas cuentas que controla, especificando la clave publica y privada de cada una.

Para cada cuenta se debera visualizar el balance de la misma, es decir la suma de todas sus UTXO disponibles al momento.

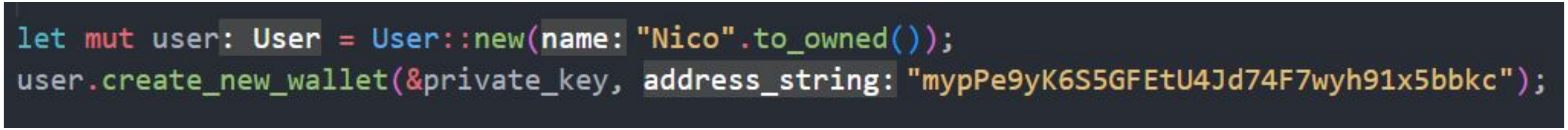


```
1 implementation
pub struct Wallet {
    pub address: String,
    pub private_key: SecretKey,
    pub public_key: secp256k1::PublicKey,
    pub utxo_set: UTXOSet,
    pub balance: u64,
    pub transactions_history: Vec<Transaction>,
    pub recieved_transactions: Vec<Transaction>,
    pub sent_transactions: Vec<Transaction>,
```

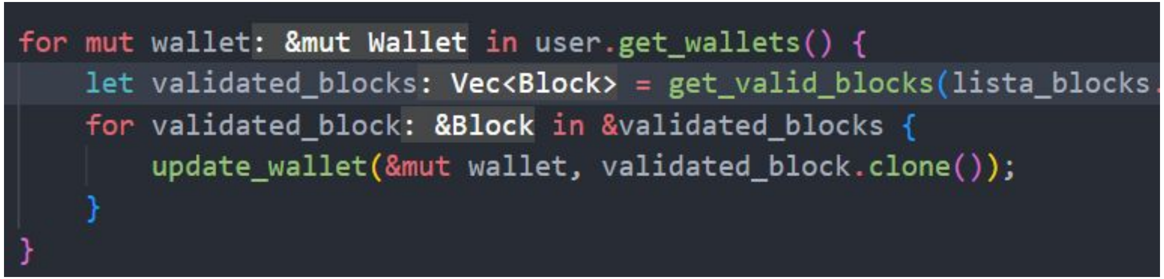


```
pub fn create_new_wallet(&mut self,
    private_key: &secp256k1::SecretKey,
    address_string: &str) {
    let wallet: Wallet = Wallet::new_from_existing(
        &private_key,
        &private_key.public_key(secp: &Secp256k1::new()),
        address_string,
    );

    self.wallets.push(wallet);
}
```

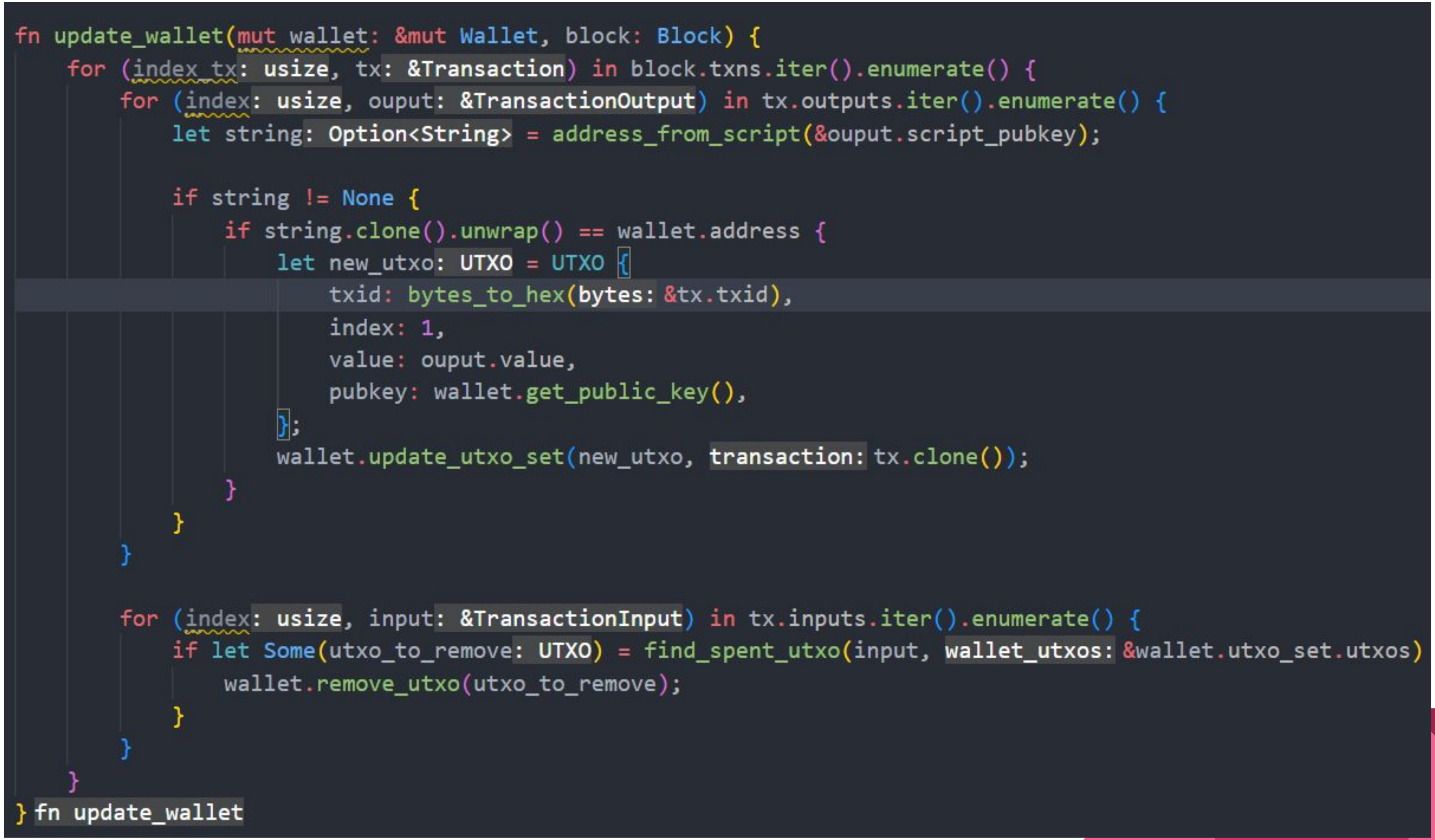
Flujo de manejo de una Wallet



```
let mut user: User = User::new(name: "Nico".to_owned());
user.create_new_wallet(&private_key, address_string: "mypPe9yK6S5GFEtU4Jd74F7wyh91x5bbkc");
```

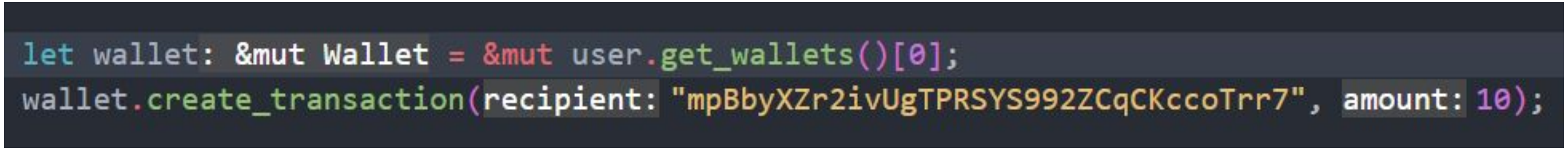


```
for mut wallet: &mut Wallet in user.get_wallets() {
    let validated_blocks: Vec<Block> = get_valid_blocks(lista_blocks.
    for validated_block: &Block in &validated_blocks {
        update_wallet(&mut wallet, validated_block.clone());
    }
}
```



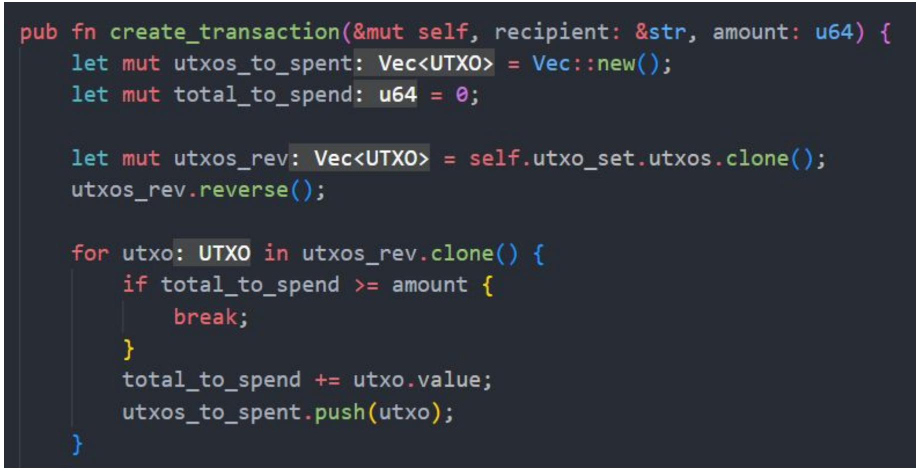
```rust
fn update_wallet(mut wallet: &mut Wallet, block: Block) {
    for (index_tx: usize, tx: &Transaction) in block.txns.iter().enumerate() {
        for (index: usize, ouput: &TransactionOutput) in tx.outputs.iter().enumerate() {
            let string: Option<String> = address_from_script(&ouput.script_pubkey);

            if string != None {
                if string.clone().unwrap() == wallet.address {
                    let new_utxo: UTXO = UTXO {
                        txid: bytes_to_hex(bytes: &tx.txid),
                        index: 1,
                        value: ouput.value,
                        pubkey: wallet.get_public_key(),
                    };
                    wallet.update_utxo_set(new_utxo, transaction: tx.clone());
                }
            }
        }

        for (index: usize, input: &TransactionInput) in tx.inputs.iter().enumerate() {
            if let Some(utxo_to_remove: UTXO) = find_spent_utxo(input, wallet_utxos: &wallet.utxo_set.utxos)
                wallet.remove_utxo(utxo_to_remove);
            }
        }
    }
} fn update_wallet
```

Flujo de Creación de una transacción



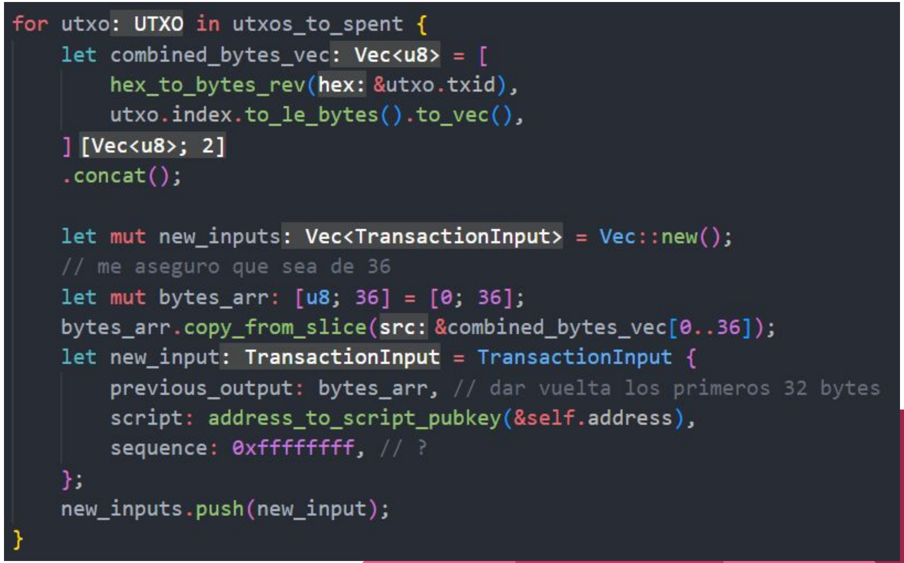
```
let wallet: &mut Wallet = &mut user.get_wallets()[0];
wallet.create_transaction(recipient: "mpBbyXZr2ivUgTPRSYS992ZCqCKccoTrr7", amount: 10);
```

1 - Se eligen los UTXOS necesarios para realizar la transacción



```
pub fn create_transaction(&mut self, recipient: &str, amount: u64) {
    let mut utxos_to_spent: Vec<UTXO> = Vec::new();
    let mut total_to_spend: u64 = 0;

    let mut utxos_rev: Vec<UTXO> = self.utxo_set.utxos.clone();
    utxos_rev.reverse();

    for utxo: UTXO in utxos_rev.clone() {
        if total_to_spend >= amount {
            break;
        }
        total_to_spend += utxo.value;
        utxos_to_spent.push(utxo);
    }
```

2 - Se crea Input por cada UTXO elegido para la gastar

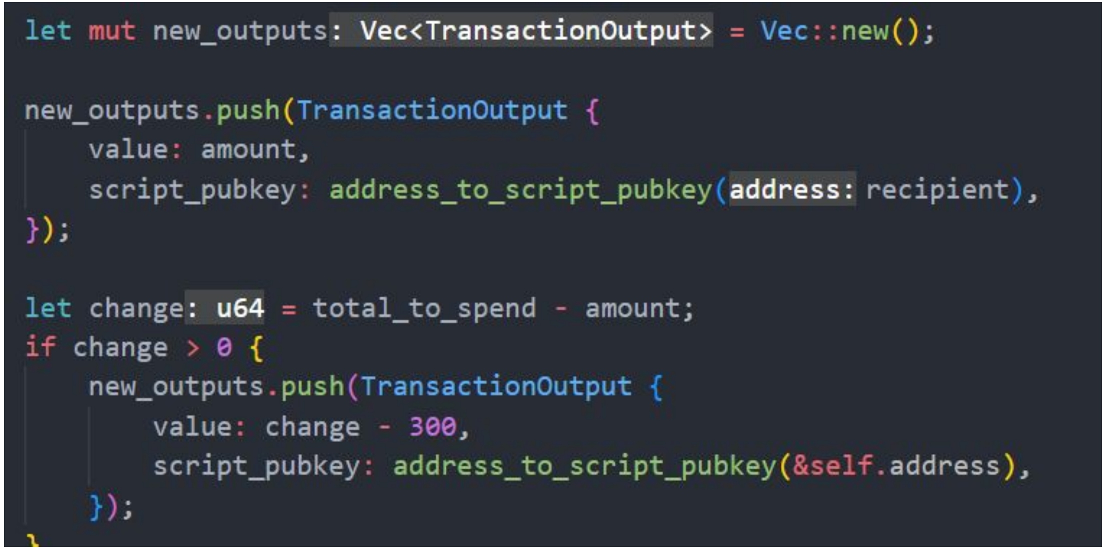


```
for utxo: UTXO in utxos_to_spent {
    let combined_bytes_vec: Vec<u8> = [
        hex_to_bytes_rev(hex: &utxo.txid),
        utxo.index.to_le_bytes().to_vec(),
    ] [Vec<u8>; 2]
    .concat();

    let mut new_inputs: Vec<TransactionInput> = Vec::new();
    // me aseguro que sea de 36
    let mut bytes_arr: [u8; 36] = [0; 36];
    bytes_arr.copy_from_slice(src: &combined_bytes_vec[0..36]);
    let new_input: TransactionInput = TransactionInput {
        previous_output: bytes_arr, // dar vuelta los primeros 32 bytes
        script: address_to_script_pubkey(&self.address),
        sequence: 0xffffffff, // ?
    };
    new_inputs.push(new_input);
}
```

Flujo de Creación de una transacción

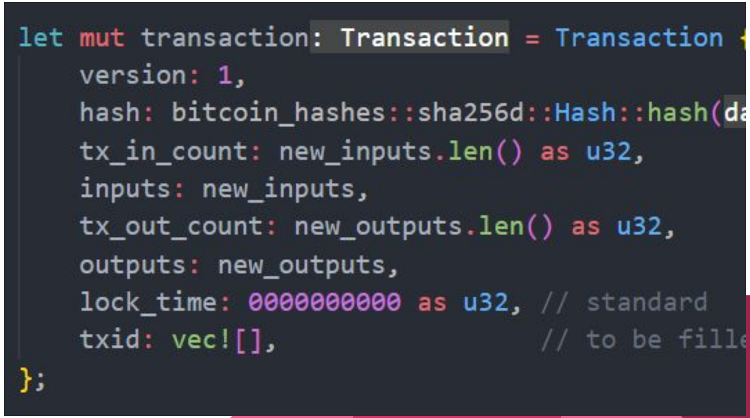
3 -Se crea Output para enviar al destinatario y se crea Output del vuelto

4 - Se elimina el output viejo y se agrega el output nuevo en nuestra lista de utxos



```
let mut new_outputs: Vec<TransactionOutput> = Vec::new();

new_outputs.push(TransactionOutput {
    value: amount,
    script_pubkey: address_to_script_pubkey(address: recipient),
});

let change: u64 = total_to_spend - amount;
if change > 0 {
    new_outputs.push(TransactionOutput {
        value: change - 300,
        script_pubkey: address_to_script_pubkey(&self.address),
    });
}
```

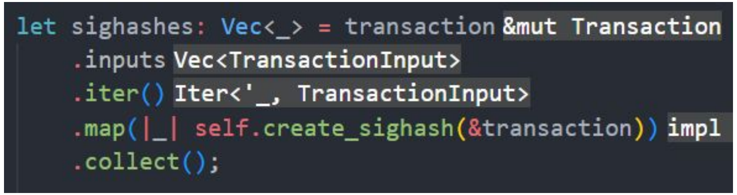
5 - Se actualiza la transacción



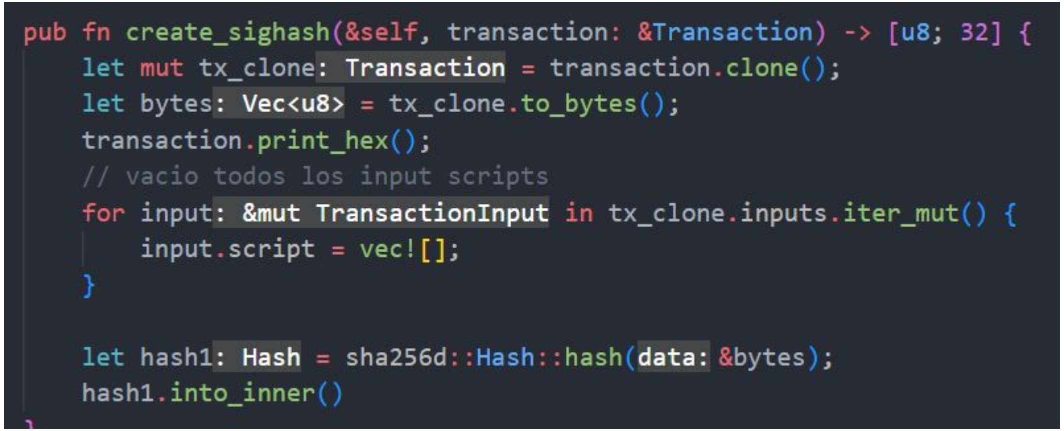
```
let mut transaction: Transaction = Transaction {
    version: 1,
    hash: bitcoin_hashes::sha256d::Hash::hash(da
    tx_in_count: new_inputs.len() as u32,
    inputs: new_inputs,
    tx_out_count: new_outputs.len() as u32,
    outputs: new_outputs,
    lock_time: 0000000000 as u32, // standard
    txid: vec![],                 // to be fille
};
```

Flujo de Creación de una transacción

6 . Se hashea dos veces la transaccion por cada input



```
let sighashes: Vec<_> = transaction &mut Transaction
    .inputs Vec<TransactionInput>
    .iter() Iter<'_, TransactionInput>
    .map(|_| self.create_sighash(&transaction)) impl
    .collect();
```

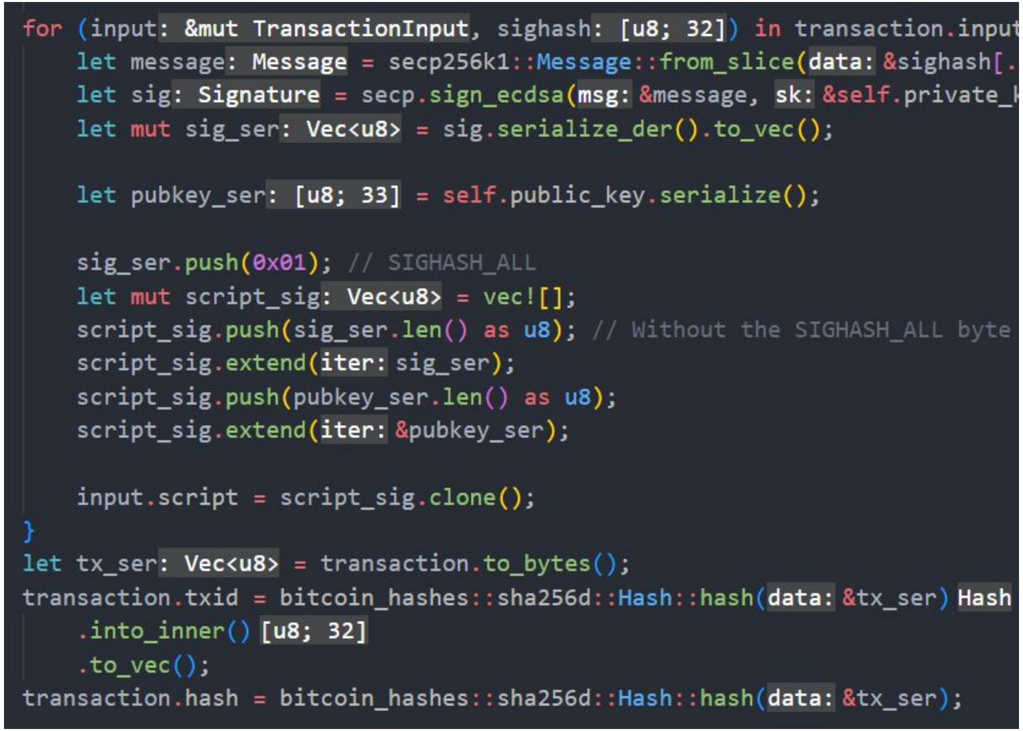


```
pub fn create_sighash(&self, transaction: &Transaction) -> [u8; 32] {
    let mut tx_clone: Transaction = transaction.clone();
    let bytes: Vec<u8> = tx_clone.to_bytes();
    transaction.print_hex();
    // vacio todos los input scripts
    for input: &mut TransactionInput in tx_clone.inputs.iter_mut() {
        input.script = vec![];
    }

    let hash1: Hash = sha256d::Hash::hash(data: &bytes);
    hash1.into_inner()
}
```

7 . Se vacian todos los inputs de la transaccion

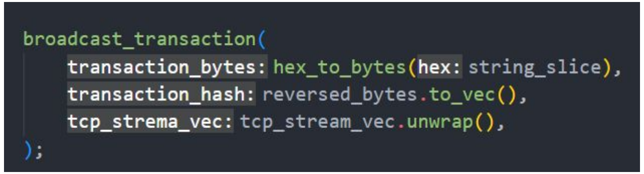
Flujo de Creación de una transacción

8 . Por cada input asignado a un hash se crea la firma y se remplaza en el input con otros campos

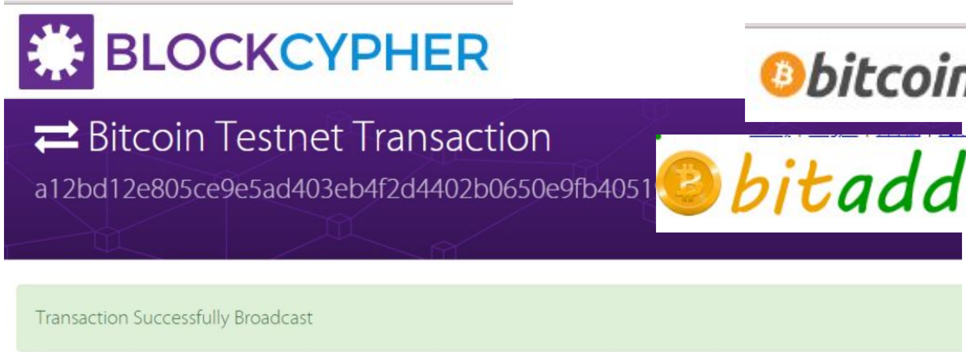


```
for (input: &mut TransactionInput, sighash: [u8; 32]) in transaction.input
    let message: Message = secp256k1::Message::from_slice(data: &sighash[.
    let sig: Signature = secp.sign_ecdsa(msg: &message, sk: &self.private_k
    let mut sig_ser: Vec<u8> = sig.serialize_der().to_vec();

    let pubkey_ser: [u8; 33] = self.public_key.serialize();

    sig_ser.push(0x01); // SIGHASH_ALL
    let mut script_sig: Vec<u8> = vec![];
    script_sig.push(sig_ser.len() as u8); // Without the SIGHASH_ALL byte
    script_sig.extend(iter: sig_ser);
    script_sig.push(pubkey_ser.len() as u8);
    script_sig.extend(iter: &pubkey_ser);

    input.script = script_sig.clone();
}
let tx_ser: Vec<u8> = transaction.to_bytes();
transaction.txid = bitcoin_hashes::sha256d::Hash::hash(data: &tx_ser) Hash
    .into_inner() [u8; 32]
    .to_vec();
transaction.hash = bitcoin_hashes::sha256d::Hash::hash(data: &tx_ser);
```

9 - Se hashea nuevamente la transaccion firmada y se broadcastea



```
broadcast_transaction(
    transaction_bytes: hex_to_bytes(hex: string_slice),
    transaction_hash: reversed_bytes.to_vec(),
    tcp_strema_vec: tcp_stream_vec.unwrap(),
);
```



BLOCKCYPHER

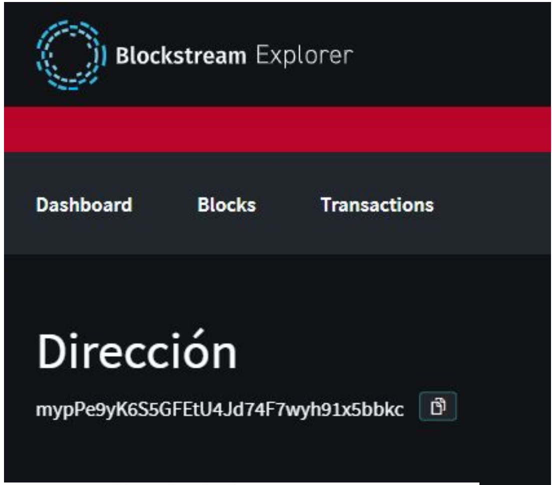
⇄ Bitcoin Testnet Transaction

a12bd12e805ce9e5ad403eb4f2d4402b0650e9fb4051

Transaction Successfully Broadcast

bitcoindeveloper

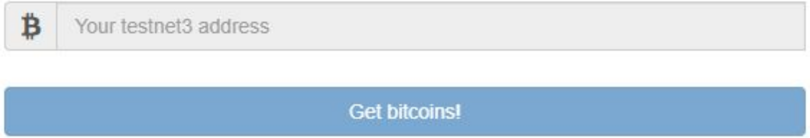
bitaddress.org

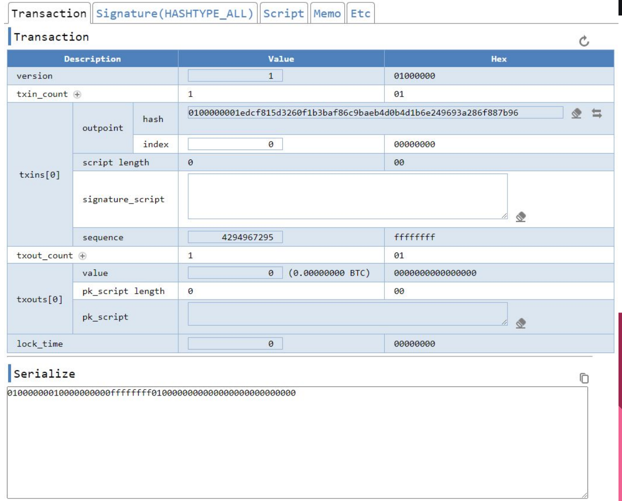


Blockstream Explorer

Dirección

mypPe9yK6S5GFEtU4Jd74F7wyh91x5bbkc



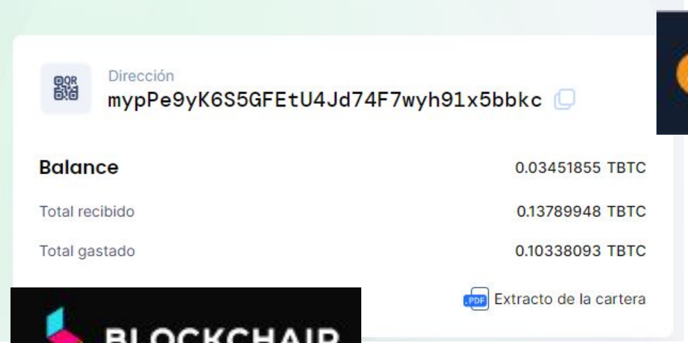


Transaction

Description			Value	Hex
version			1	01000000
txin_count ⊕			1	01
txins[0]	outpoint	hash	0100000001edcf815d3260f1b3baf86c9baeb4d0b4d1b6e249693a286f887b96	
		index	0	00000000
	script length		0	00
	signature_script			
	sequence		4294967295	ffffffff
txout_count ⊕			1	01
txouts[0]	value		0 (0.00000000 BTC)	0000000000000000
	pk_script length		0	00
	pk_script			
lock_time			0	00000000

Serialize

```
01000000010000000000ffffffff010000000000000000000000000000
```



Dirección

mypPe9yK6S5GFEtU4Jd74F7wyh91x5bbkc

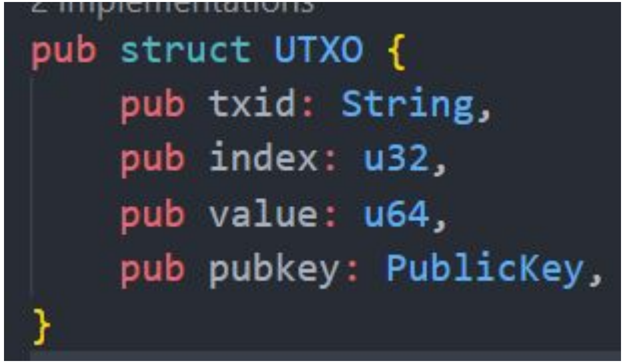
Balance	0.03451855 TBTC
Total recibido	0.13789948 TBTC
Total gastado	0.10338093 TBTC

Extracto de la cartera

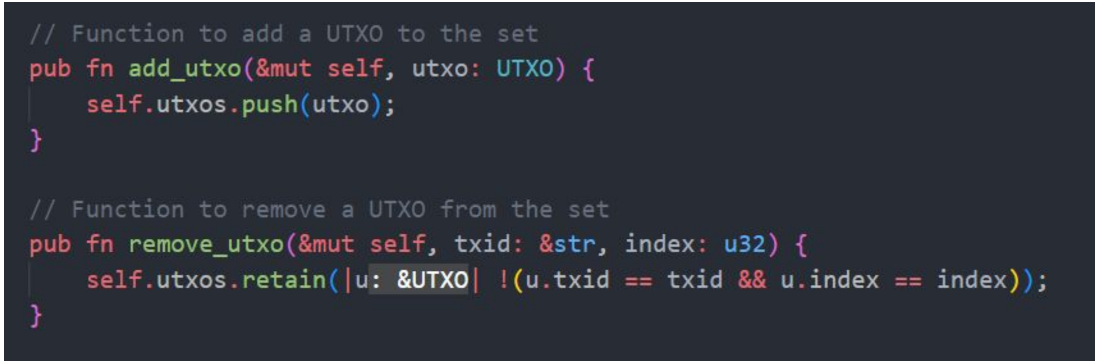
bitcoin testnet

BLOCKCHAIR

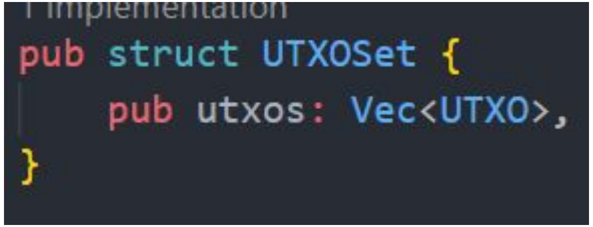
UtxoSet



```
2 implementations
pub struct UTXO {
    pub txid: String,
    pub index: u32,
    pub value: u64,
    pub pubkey: PublicKey,
}
```



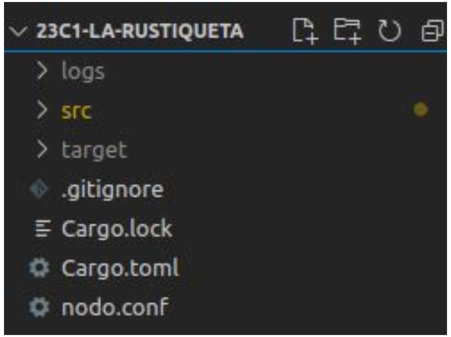
```
// Function to add a UTXO to the set
pub fn add_utxo(&mut self, utxo: UTXO) {
    self.utxos.push(utxo);
}

// Function to remove a UTXO from the set
pub fn remove_utxo(&mut self, txid: &str, index: u32) {
    self.utxos.retain(|u: &UTXO| !(u.txid == txid && u.index == index));
}
```



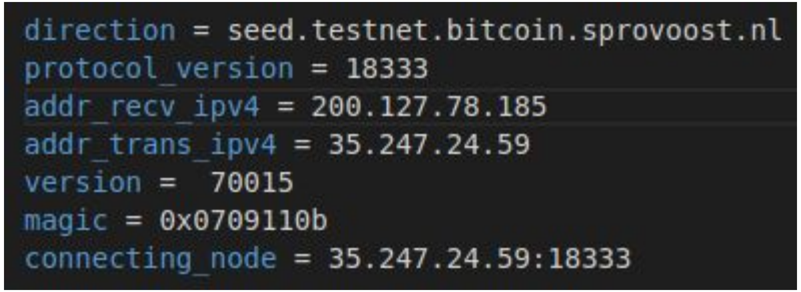
```
1 implementation
pub struct UTXOSet {
    pub utxos: Vec<UTXO>,
}
```

Archivo de configuración

Definimos el formato del archivo de configuración de la siguiente manera



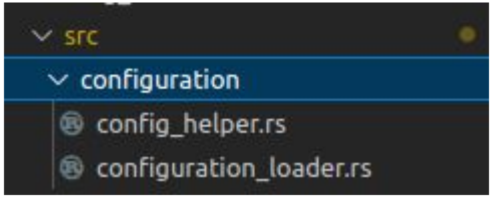
```
23C1-LA-RUSTIQUETA
  > logs
  > src
  > target
  .gitignore
  Cargo.lock
  Cargo.toml
  nodo.conf
```

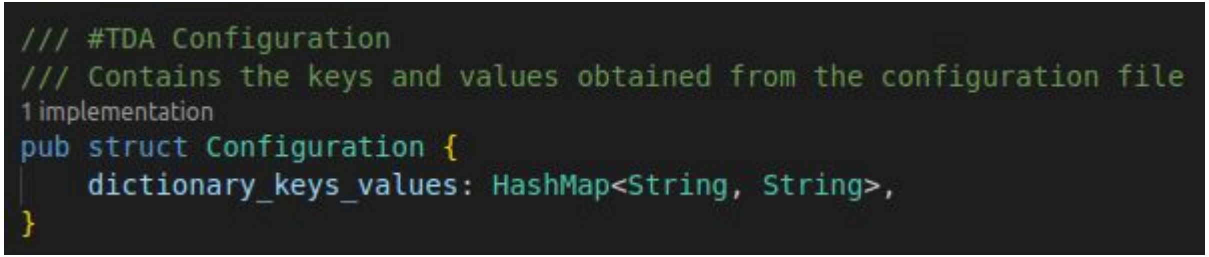


```
direction = seed.testnet.bitcoin.sprovoost.nl
protocol_version = 18333
addr_recv_ipv4 = 200.127.78.185
addr_trans_ipv4 = 35.247.24.59
version =  70015
magic = 0x0709110b
connecting_node = 35.247.24.59:18333
```

Archivo de configuración

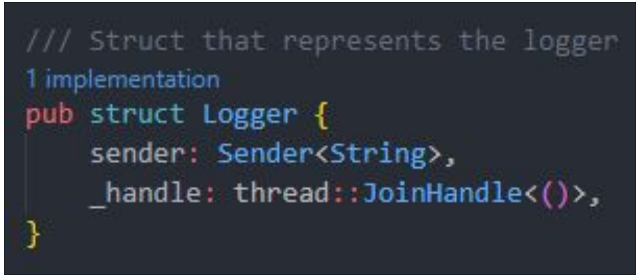
Modelado



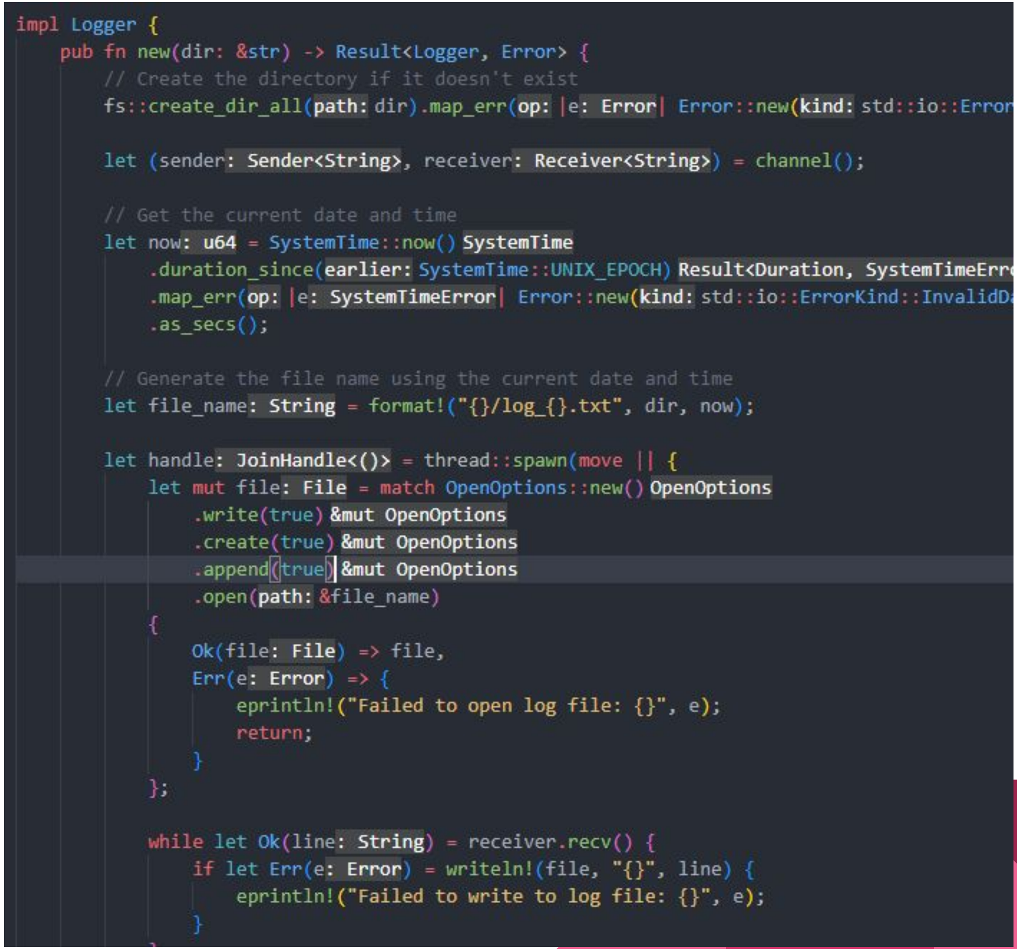


```
/// #TDA Configuration
/// Contains the keys and values obtained from the configuration file
1 implementation
pub struct Configuration {
    dictionary_keys_values: HashMap<String, String>,
}
```

Log File



```
/// Struct that represents the logger
1 implementation
pub struct Logger {
    sender: Sender<String>,
    _handle: thread::JoinHandle<()>,
}
```



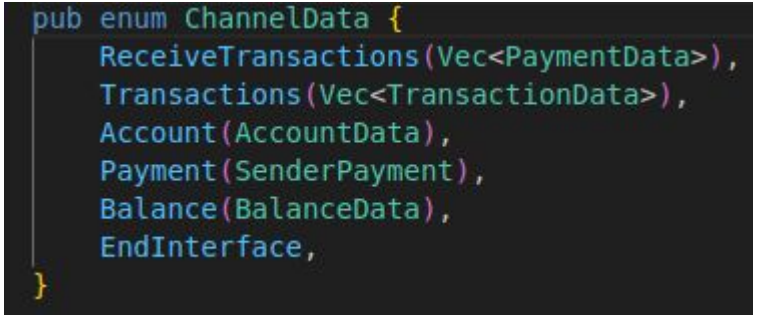
```
impl Logger {
    pub fn new(dir: &str) -> Result<Logger, Error> {
        // Create the directory if it doesn't exist
        fs::create_dir_all(path: dir).map_err(op: |e: Error| Error::new(kind: std::io::Error

        let (sender: Sender<String>, receiver: Receiver<String>) = channel();

        // Get the current date and time
        let now: u64 = SystemTime::now() SystemTime
            .duration_since(earlier: SystemTime::UNIX_EPOCH) Result<Duration, SystemTimeErr
            .map_err(op: |e: SystemTimeError| Error::new(kind: std::io::ErrorKind::InvalidD
            .as_secs();

        // Generate the file name using the current date and time
        let file_name: String = format!("{}/log_{}.txt", dir, now);

        let handle: JoinHandle<()> = thread::spawn(move || {
            let mut file: File = match OpenOptions::new() OpenOptions
                .write(true) &mut OpenOptions
                .create(true) &mut OpenOptions
                .append(true) &mut OpenOptions
                .open(path: &file_name)
            {
                Ok(file: File) => file,
                Err(e: Error) => {
                    eprintln!("Failed to open log file: {}", e);
                    return;
                }
            };

            while let Ok(line: String) = receiver.recv() {
                if let Err(e: Error) = writeln!(file, "{}", line) {
                    eprintln!("Failed to write to log file: {}", e);
                }
```

Utilización de hilos

- Hilo para interfaz
- Hilo para comportamiento de nodo

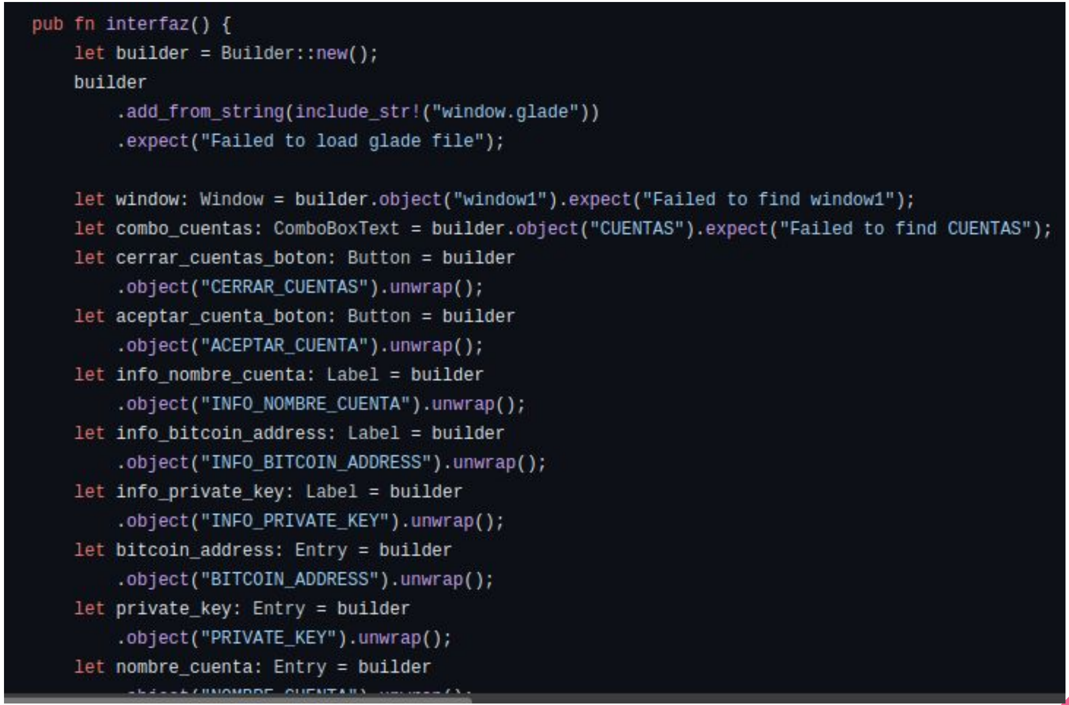
Comunicación entre nodo e interfaz

- Channel para envío de datos de nodo a interfaz GLIB
- Channel para envío de datos de interfaz a nodo MPSC



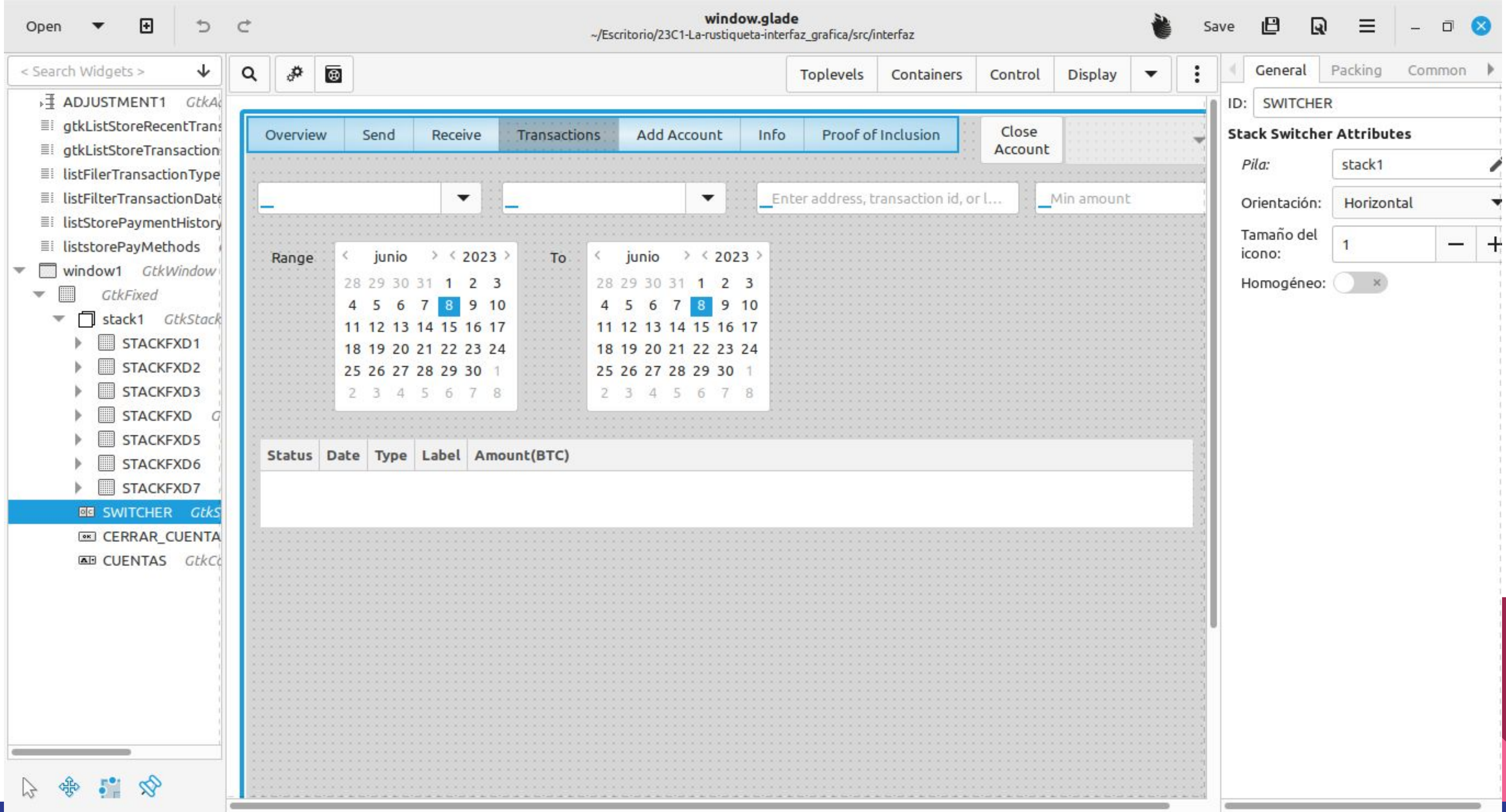
```
pub enum ChannelData {
    ReceiveTransactions(Vec<PaymentData>),
    Transactions(Vec<TransactionData>),
    Account(AccountData),
    Payment(SenderPayment),
    Balance(BalanceData),
    EndInterface,
}
```

GTK & Glade



```
pub fn interfaz() {
    let builder = Builder::new();
    builder
        .add_from_string(include_str!("window.glade"))
        .expect("Failed to load glade file");

    let window: Window = builder.object("window1").expect("Failed to find window1");
    let combo_cuentas: ComboBoxText = builder.object("CUENTAS").expect("Failed to find CUENTAS");
    let cerrar_cuentas_boton: Button = builder
        .object("CERRAR_CUENTAS").unwrap();
    let aceptar_cuenta_boton: Button = builder
        .object("ACEPTAR_CUENTA").unwrap();
    let info_nombre_cuenta: Label = builder
        .object("INFO_NOMBRE_CUENTA").unwrap();
    let info_bitcoin_address: Label = builder
        .object("INFO_BITCOIN_ADDRESS").unwrap();
    let info_private_key: Label = builder
        .object("INFO_PRIVATE_KEY").unwrap();
    let bitcoin_address: Entry = builder
        .object("BITCOIN_ADDRESS").unwrap();
    let private_key: Entry = builder
        .object("PRIVATE_KEY").unwrap();
    let nombre_cuenta: Entry = builder
        .object("NOMBRE_CUENTA").unwrap();
```

GTK & Glade



window.glade
~/Escritorio/23C1-La-rustiqueta-interfaz_grafica/src/interfaz

ADJUSTMENT1 GtkA
gtkListStoreRecentTrans
gtkListStoreTransaction
listFilerTransactionType
listFilterTransactionDate
listStorePaymentHistory
liststorePayMethods
window1 GtkWindow
GtkFixed
stack1 GtkStack
STACKFXD1
STACKFXD2
STACKFXD3
STACKFXD
STACKFXD5
STACKFXD6
STACKFXD7
SWITCHER GtkS
CERRAR_CUENTA
CUENTAS GtkC

ID: SWITCHER

Stack Switcher Attributes

Pila: stack1

Orientación: Horizontal

Tamaño del icono: 1

Homogéneo:

Forma de trabajo

- Git
- Issues
- Pull Request (aprobar 2 personas)
- Documentación y funciones en inglés
- Cargo clippy y Cargo fmt
- Testing

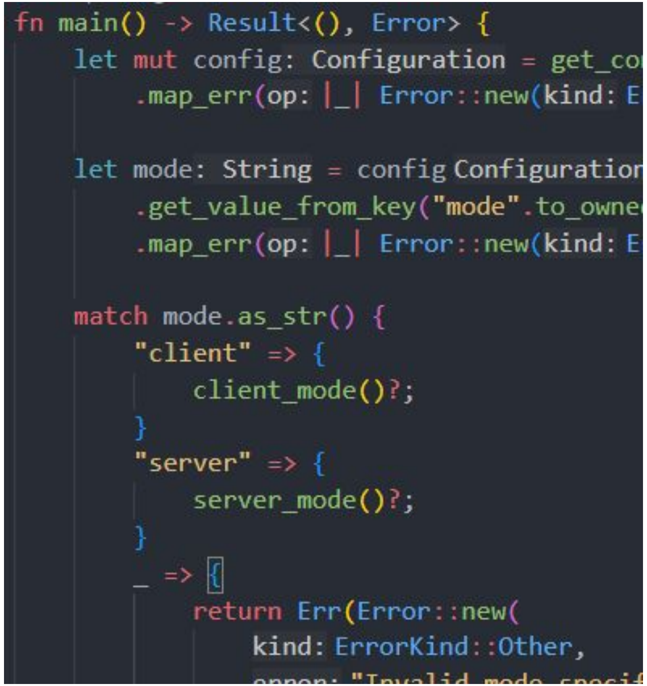
TP BITCOIN
Entrega Final

Integrantes:

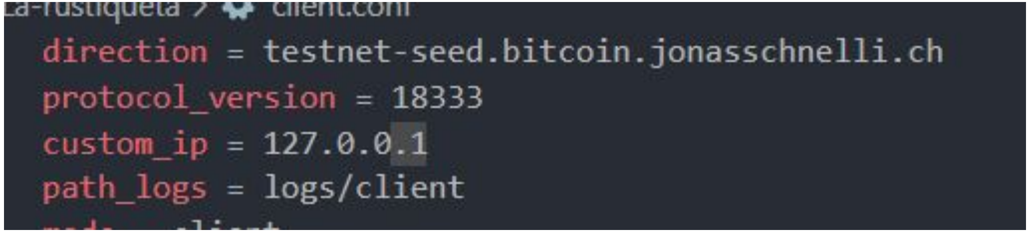
- Fabián Colque
- Nicolas Vagó
- Sofia Marchesini
- Sebastian Makkos

Modo Cliente y Modo Servidor - Modos de Conexión

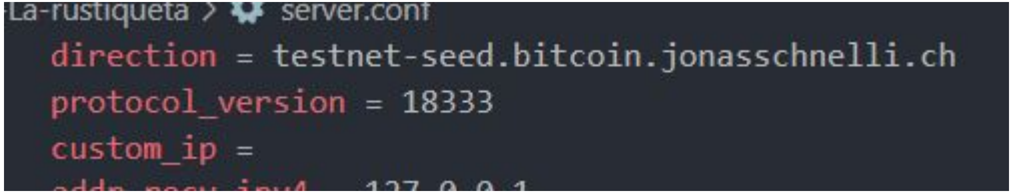
- Dos configuraciones que se corren en dos terminales distintas



```
fn main() -> Result<(), Error> {
    let mut config: Configuration = get_co
        .map_err(op: |_| Error::new(kind: E

    let mode: String = config Configuration
        .get_value_from_key("mode".to_owne
        .map_err(op: |_| Error::new(kind: E

    match mode.as_str() {
        "client" => {
            client_mode()?;
        }
        "server" => {
            server_mode()?;
        }
        _ => {
            return Err(Error::new(
                kind: ErrorKind::Other,
                error: "Invalid mode specif
```

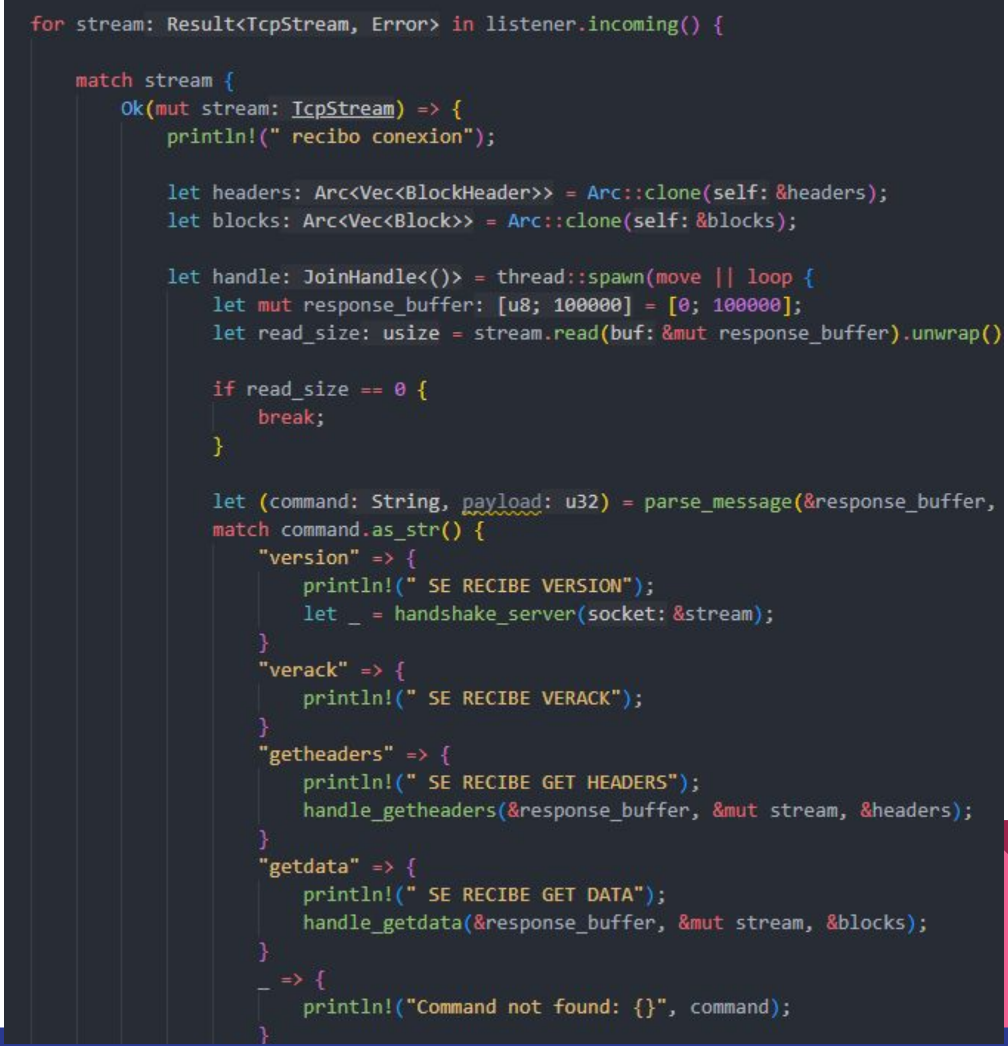


```
direction = testnet-seed.bitcoin.jonasschnelli.ch
protocol_version = 18333
custom_ip = 127.0.0.1
path_logs = logs/client
mode = client
```



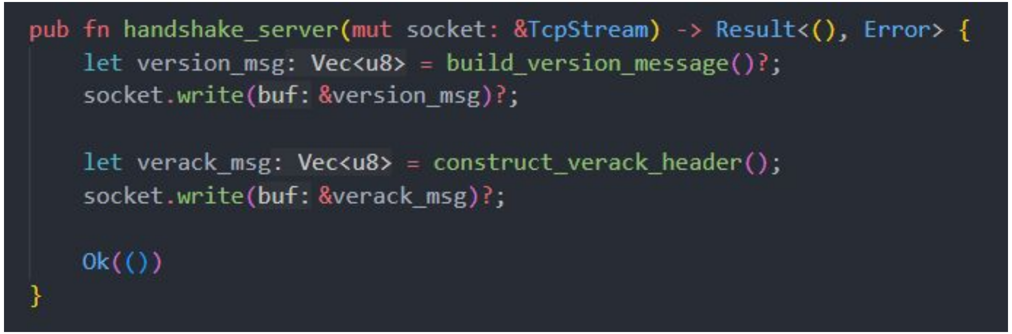
```
direction = testnet-seed.bitcoin.jonasschnelli.ch
protocol_version = 18333
custom_ip =
addr_recv_ipv4 = 127.0.0.1
```

Modo Cliente y Modo Servidor - Server Response

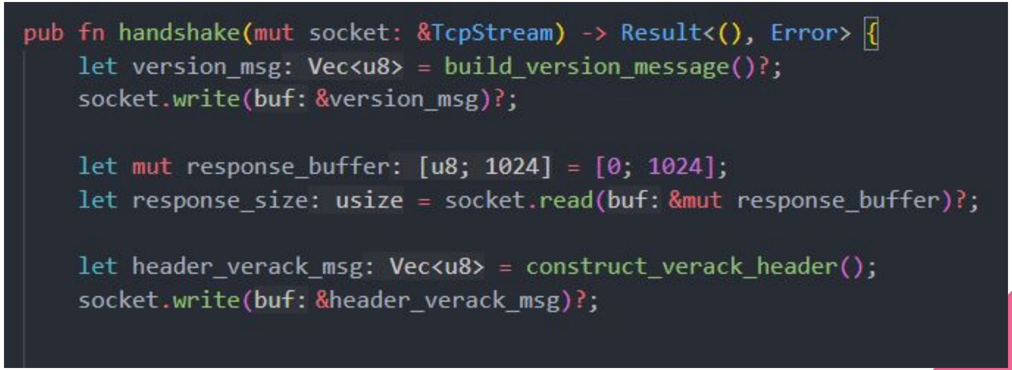


```
for stream: Result<TcpStream, Error> in listener.incoming() {

    match stream {
        Ok(mut stream: TcpStream) => {
            println!(" recibo conexion");

            let headers: Arc<Vec<BlockHeader>> = Arc::clone(self: &headers);
            let blocks: Arc<Vec<Block>> = Arc::clone(self: &blocks);

            let handle: JoinHandle<()> = thread::spawn(move || loop {
                let mut response_buffer: [u8; 100000] = [0; 100000];
                let read_size: usize = stream.read(buf: &mut response_buffer).unwrap()

                if read_size == 0 {
                    break;
                }

                let (command: String, payload: u32) = parse_message(&response_buffer,
                match command.as_str() {
                    "version" => {
                        println!(" SE RECIBE VERSION");
                        let _ = handshake_server(socket: &stream);
                    }
                    "verack" => {
                        println!(" SE RECIBE VERACK");
                    }
                    "getheaders" => {
                        println!(" SE RECIBE GET HEADERS");
                        handle_getheaders(&response_buffer, &mut stream, &headers);
                    }
                    "getdata" => {
                        println!(" SE RECIBE GET DATA");
                        handle_getdata(&response_buffer, &mut stream, &blocks);
                    }
                    _ => {
                        println!("Command not found: {}", command);
                    }
```

Modo Cliente y Modo Servidor - Conexión

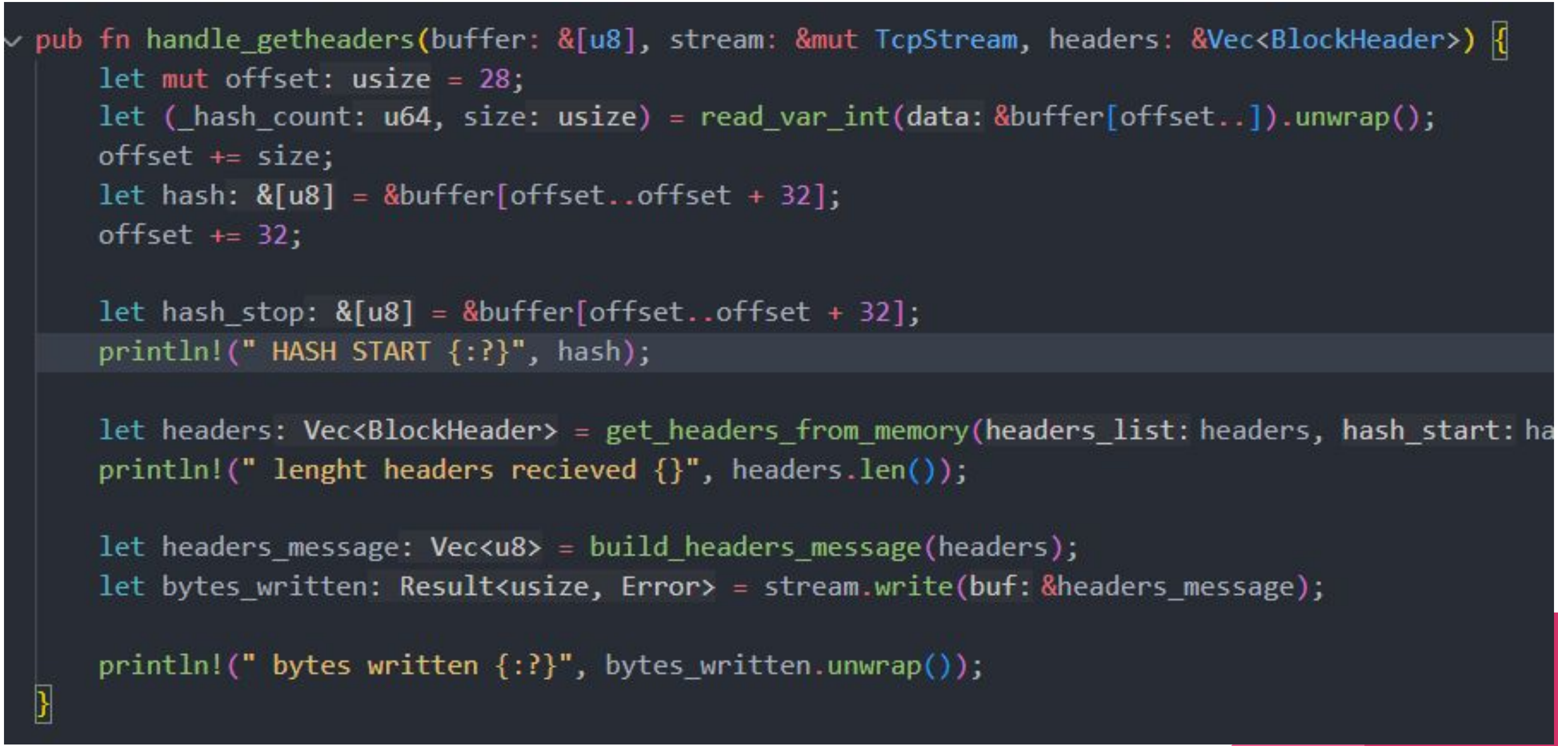


```rust
pub fn handshake_server(mut socket: &TcpStream) -> Result<(), Error> {
    let version_msg: Vec<u8> = build_version_message()?;
    socket.write(buf: &version_msg)?;

    let verack_msg: Vec<u8> = construct_verack_header();
    socket.write(buf: &verack_msg)?;

    Ok(())
}
```



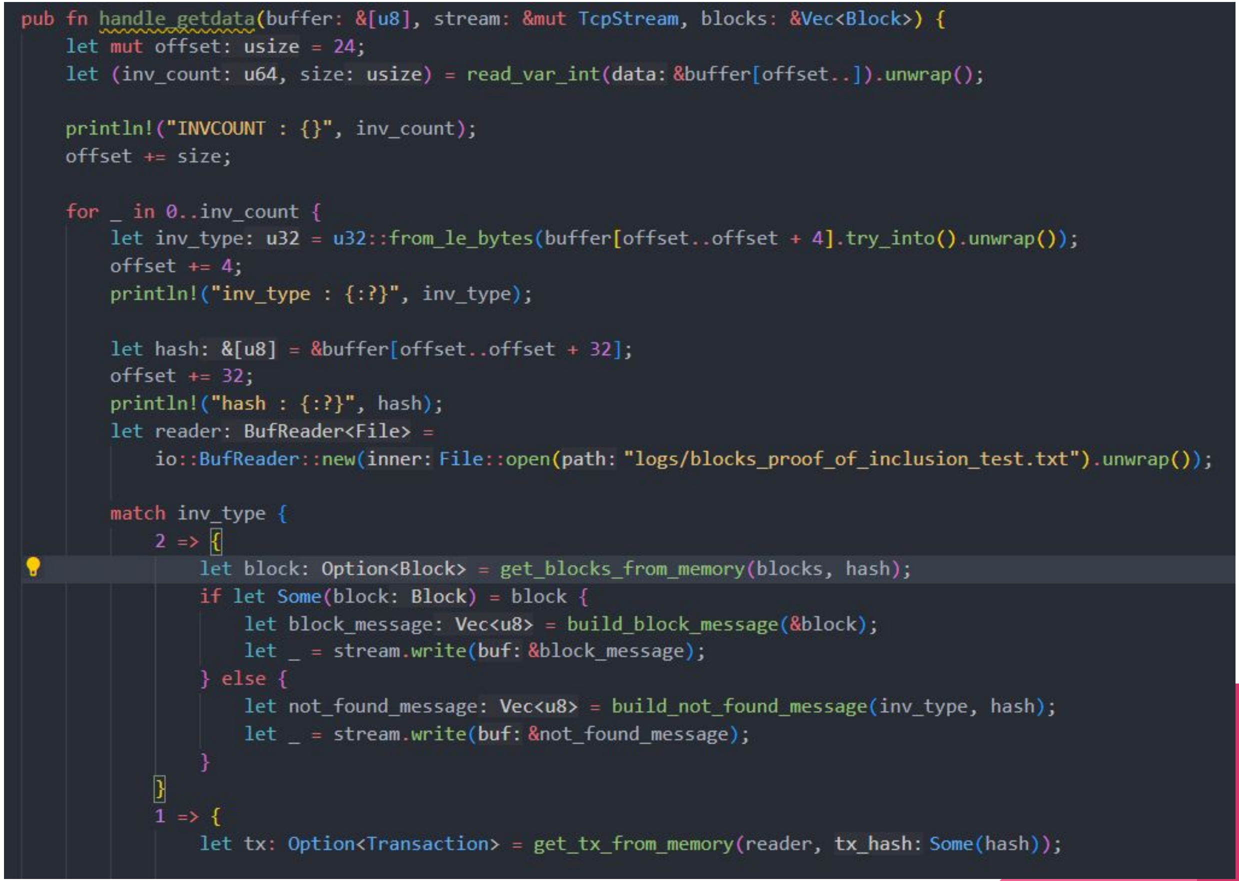
```rust
pub fn handshake(mut socket: &TcpStream) -> Result<(), Error> {
    let version_msg: Vec<u8> = build_version_message()?;
    socket.write(buf: &version_msg)?;

    let mut response_buffer: [u8; 1024] = [0; 1024];
    let response_size: usize = socket.read(buf: &mut response_buffer)?;

    let header_verack_msg: Vec<u8> = construct_verack_header();
    socket.write(buf: &header_verack_msg)?;
```

Modo Cliente y Modo Servidor - GetHeaders Response



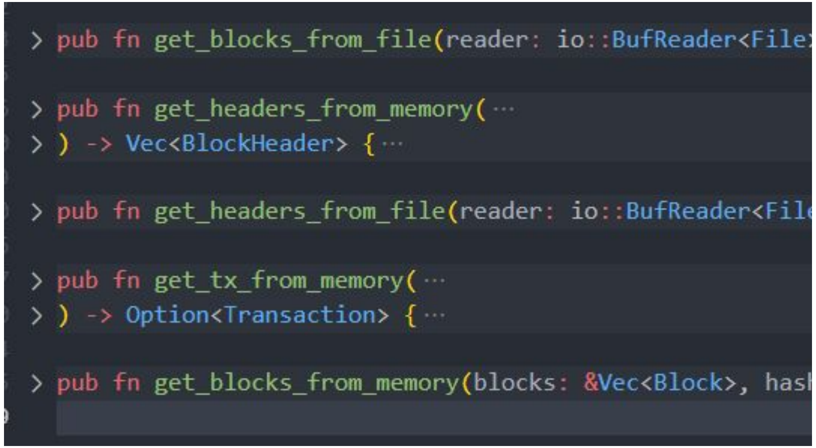
```
pub fn handle_getheaders(buffer: &[u8], stream: &mut TcpStream, headers: &Vec<BlockHeader>) {
    let mut offset: usize = 28;
    let (_hash_count: u64, size: usize) = read_var_int(data: &buffer[offset..]).unwrap();
    offset += size;
    let hash: &[u8] = &buffer[offset..offset + 32];
    offset += 32;

    let hash_stop: &[u8] = &buffer[offset..offset + 32];
    println!(" HASH START {:?}", hash);

    let headers: Vec<BlockHeader> = get_headers_from_memory(headers_list: headers, hash_start: ha
    println!(" lenght headers recieved {}", headers.len());

    let headers_message: Vec<u8> = build_headers_message(headers);
    let bytes_written: Result<usize, Error> = stream.write(buf: &headers_message);

    println!(" bytes written {:?}", bytes_written.unwrap());
}
```

Modo Cliente y Modo Servidor - GetData Response

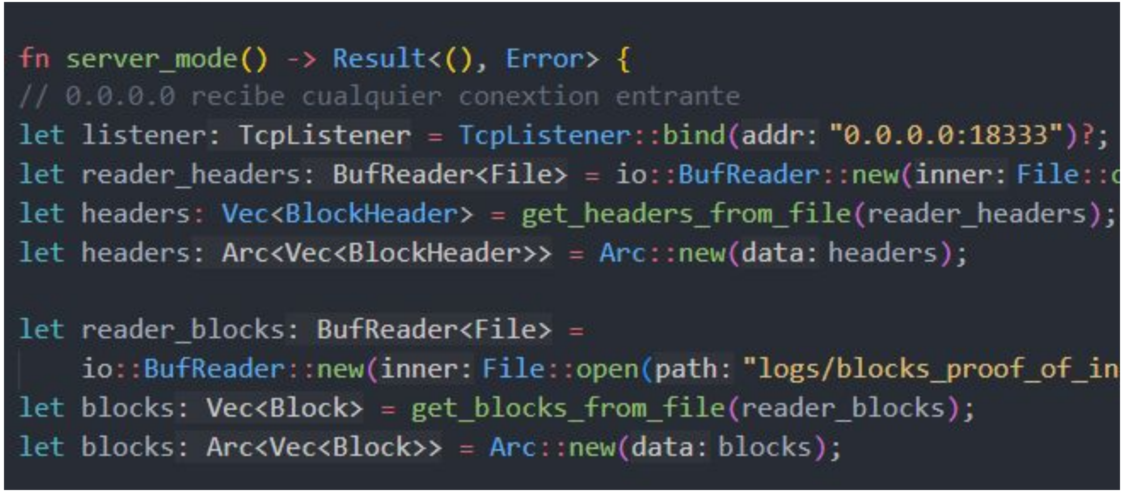


```
pub fn handle_getdata(buffer: &[u8], stream: &mut TcpStream, blocks: &Vec<Block>) {
    let mut offset: usize = 24;
    let (inv_count: u64, size: usize) = read_var_int(data: &buffer[offset..]).unwrap();

    println!("INVCOUNT : {}", inv_count);
    offset += size;

    for _ in 0..inv_count {
        let inv_type: u32 = u32::from_le_bytes(buffer[offset..offset + 4].try_into().unwrap());
        offset += 4;
        println!("inv_type : {:?}", inv_type);

        let hash: &[u8] = &buffer[offset..offset + 32];
        offset += 32;
        println!("hash : {:?}", hash);
        let reader: BufReader<File> =
            io::BufReader::new(inner: File::open(path: "logs/blocks_proof_of_inclusion_test.txt").unwrap());

        match inv_type {
            2 => {
                let block: Option<Block> = get_blocks_from_memory(blocks, hash);
                if let Some(block: Block) = block {
                    let block_message: Vec<u8> = build_block_message(&block);
                    let _ = stream.write(buf: &block_message);
                } else {
                    let not_found_message: Vec<u8> = build_not_found_message(inv_type, hash);
                    let _ = stream.write(buf: &not_found_message);
                }
            }
            1 => {
                let tx: Option<Transaction> = get_tx_from_memory(reader, tx_hash: Some(hash));
```

Modo Cliente y Modo Servidor - Server Persistance

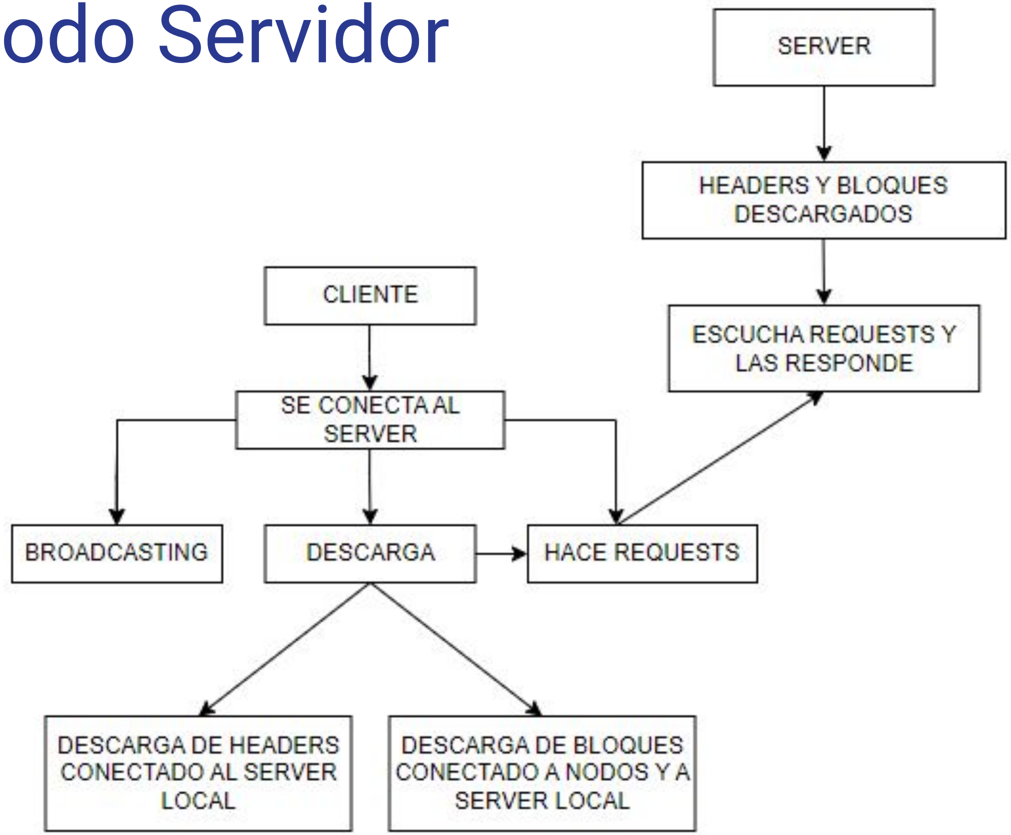


```
> pub fn get_blocks_from_file(reader: io::BufReader<File

> pub fn get_headers_from_memory(···
> ) -> Vec<BlockHeader> {···

> pub fn get_headers_from_file(reader: io::BufReader<File

> pub fn get_tx_from_memory(···
> ) -> Option<Transaction> {···

> pub fn get_blocks_from_memory(blocks: &Vec<Block>, has
```



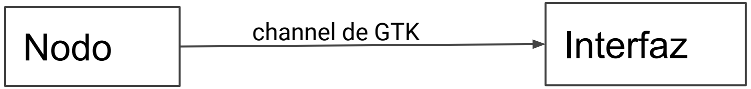
```
fn server_mode() -> Result<(), Error> {
// 0.0.0.0 recibe cualquier conextion entrante
let listener: TcpListener = TcpListener::bind(addr: "0.0.0.0:18333")?;
let reader_headers: BufReader<File> = io::BufReader::new(inner: File::c
let headers: Vec<BlockHeader> = get_headers_from_file(reader_headers);
let headers: Arc<Vec<BlockHeader>> = Arc::new(data: headers);

let reader_blocks: BufReader<File> =
    io::BufReader::new(inner: File::open(path: "logs/blocks_proof_of_in
let blocks: Vec<Block> = get_blocks_from_file(reader_blocks);
let blocks: Arc<Vec<Block>> = Arc::new(data: blocks);
```

Modo Cliente y Modo Servidor - Estructura


SERVER
HEADERS Y BLOQUES DESCARGADOS
ESCUCHA REQUESTS Y LAS RESPONDE
CLIENTE
SE CONECTA AL SERVER
BROADCASTING
DESCARGA
HACE REQUESTS
DESCARGA DE HEADERS CONECTADO AL SERVER LOCAL
DESCARGA DE BLOQUES CONECTADO A NODOS Y A SERVER LOCAL

Interfaz - Barra de Progreso

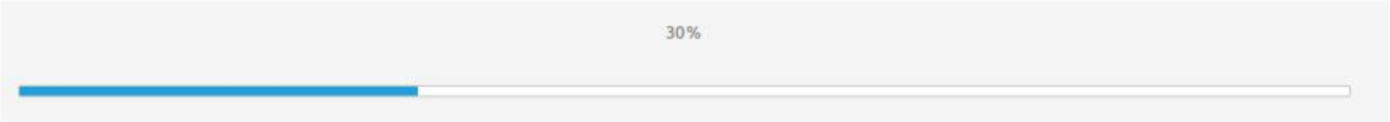
Los bloques descargados por el servidor son informados al cliente mediante un channel de GTK, el cual a medida que envía los bloques, va actualizando una barra de progreso en la interfaz del cliente.



Nodo

channel de GTK

Interfaz



30%

Presentación de Demo

Referencias

- https://developer.bitcoin.org
- https://blockstream.info/testnet/
- https://en.bitcoin.it/wiki/Protocol_documentation
- https://www.oreilly.com/library/view/programming-bitcoin/9781492031482/

¿Preguntas?