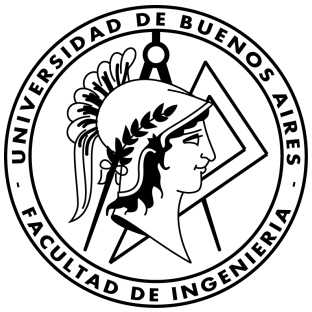


Trabajo Práctico Bitcoin — Informe Final

Taller de Programacion I
Primer cuatrimestre de 2023

Alumno	Número de padrón	Email
Masri, Noah	108814	noahmasri19@gmail.com
Ayala, Camila	107440	cayala@fi.uba.ar
Pol, Juan Manuel	108448	jpol@fi.uba.ar
Cominotti, Claudia	104891	ccominotti@fi.uba.ar

Índice

1. Introducción

Para el presente trabajo fueron pedidas una serie de tareas. Entre estas se encontraban principalmente la implementacion de un nodo bitcoin y de billeteras virtuales que se conectasen a dicho nodo, las cuales seran detalladas a continuacion.

El trabajo se basa en la estructura cliente servidor, mediante la cual el usuario, propietario de una o mas billeteras, cada vez que desea relacionarse con la Peer to Peer Network de bitcoin debe realizar pedidos al nodo, que funciona como servidor y cumple con sus pedidos siempre y cuando estos sean posibles.

2. Nodo

La estructura de nodo es el equivalente a un nodo bitcoin, y cumple todas las funciones de un nodo incompleto.

```
pub struct Node {
    pub blockchain: Blockchain,
    pub accounts: HashMap<[u8; 20], TcpStream>,
    pub broadcasted_tx: HashMap<[u8; 32], (Transaction, [u8; 20])>,
    pub peers: Vec<TcpStream>,
}

pub struct Blockchain {
    pub headers: Vec<BlockHeader>,
    pub first_block_index: usize,
    pub blocks: Vec<Block>,
    pub utxo_set: HashMap<[u8; 20], Vec<UtxoInfo>>,
}
```

Figura 1: Struct nodo.

Como se ve en dicha figura, su estructura cuenta con una blockchain propia, la cual tiene un vector de encabezados de bloques, un vector de bloques y un hashmap de Unspent Transaction Output, donde la clave es el hash de la clave publica de quien la puede gastar, y el valor es toda la informacion que este requeria para gastarlo. La eleccion de esto ultimo se debe a que esto facilita mucho la entrega de los outputs que puede gastar a una billetera que lo requiera en cualquier momento dado.

Ademas, el nodo cuenta con un hashmap de cuentas asociadas y otro de transacciones enviadas. En el primero se hace uso nuevamente del hash de la clave publica para que el pase sea immediato, y el valor es el stream mediante el cual sucede la comunicacion con este. Esto es muy util cuando se

quiere hacer un anuncio de ingreso de bloque o de transaccion que incumba al usuario. El segundo contiene las transacciones que durante su ejecucion, la billetera le pidio que cree, donde la clave es el hash de la transaccion y el valor es la transaccion misma y el hash de la clave publica del usuario. El hash se guarda con el proposito de notificar la entrada del bloque que la contenga, mientras que la transaccion se guarda para poder enviarla cada vez que se recibe un mensaje GetData de algun peer.

En ultimo lugar, guarda un vector con sus peers de la red bitcoin, mediante los cuales esta constantemente escuchando.

2.1. Creacion

El nodo se inicializa haciendo un handshake, donde se conecta a todos los peers que puede, usando las direcciones obtenidas. Luego de esto, hace uso de una thread pool llamada Block Getter especifica, mediante la cual descarga todos los bloques a partir de la fecha especificada en el archivo de configuracion.

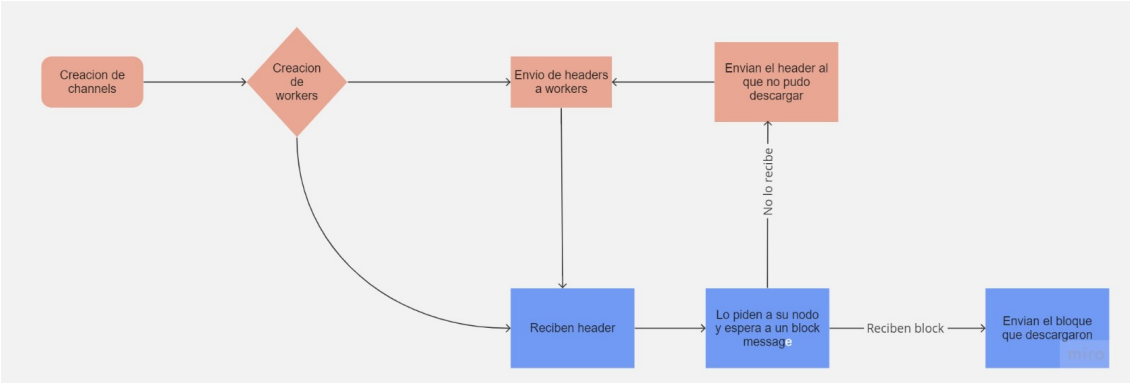
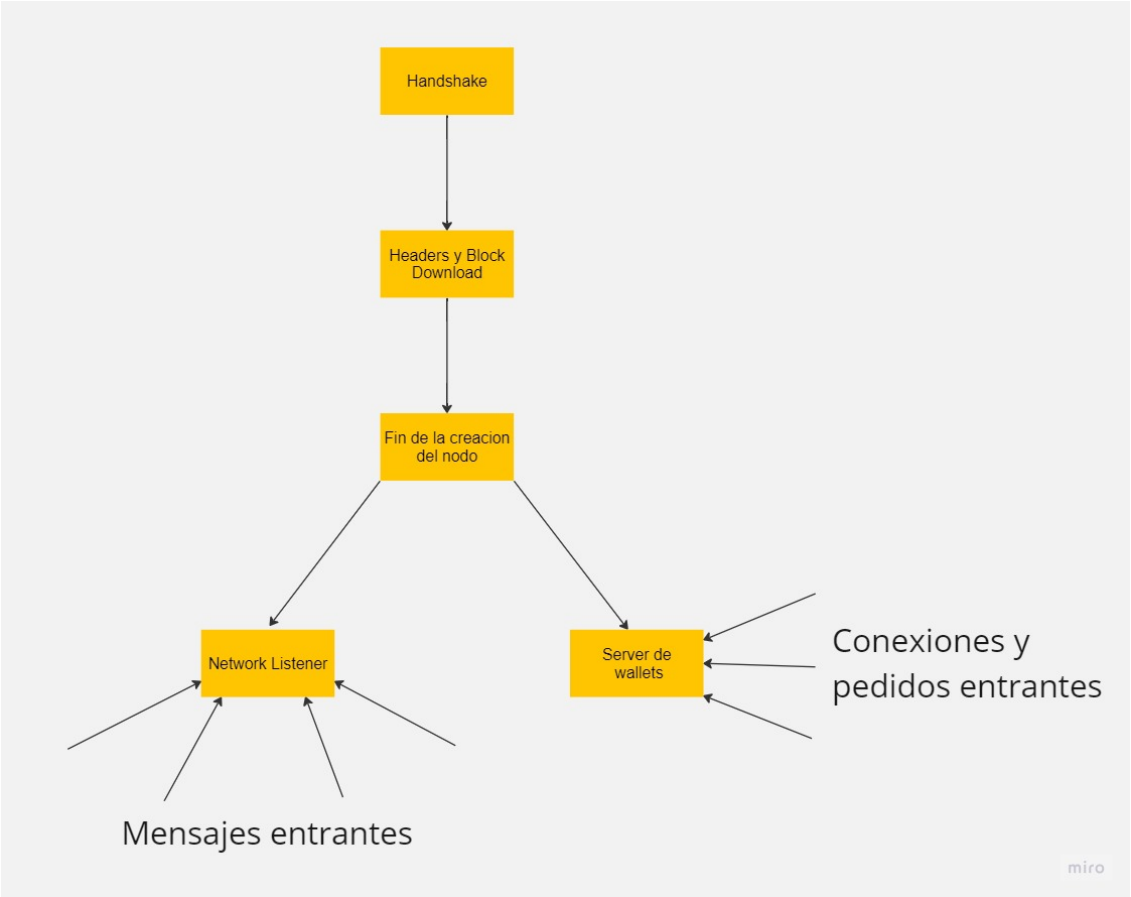


Figura 2: Thread pool de descarga de bloques.

La descarga sucede de la manera detallada en el diagrama de flujo anterior. Se inicializa creando los canales de comunicacion entre el orquestador y los trabajadores. Se crea un hilo por cada uno de los peers que tenemos, y se mandan por el hilo las tareas a realizar. Las tareas son representadas por los encabezados de los bloques que se desea que cada trabajador descargue. Si este es incapaz de realizar su tarea, sea por el motivo que fuere, se reenvia este encabezado al orquestador para que este se encargue de volver a mandarlo por el canal de tareas, esperando que lo reciba otro.



Una vez que no quedan mas encabezados, o se pasa mucho tiempo, se destruye esta estructura, se ordenan los bloques, se crea el set de UTXOs y si todo esta correcto, se da por finalizada la descarga de bloques. Una vez que sucede esto, se inicializan los otros hashmaps vacios y se continua con el programa trabajando con multiples threads.

Tendremos alli un network listener, que se encarga de escuchar los mensajes transmitidos por los peers de la red, y un handler de conexiones con las wallets.

2.2. Network Listener

Una vez finalizado el inicial blockdownload el nodo debe mantenerse conectado a sus peers y hacer nota de cambios en la blockchain. Se diseño un network listener, quien crea un worker por cada peer y lleva registro de los recibido en dos hashmaps, uno para las transacciones y uno para los bloques. Ademas cuenta con una referencia mutex al nodo.
Informacion relevante es enviada por cada worker al orquestador por medio de un channel, especificamente cuando llega un block message y cuando llega un tx message. Los mensajes pings son

respondidos con un mensaje pong y el resto (como sendheaders y feefilter) son ignorados.
El orquestador recibe los bloques anunciados y si es necesario lockea el nodo e indica que debe agregarlo a su blockchain. Ademas si el bloque contiene una transaccion en la cual se ve involucrado alguna de las cuentas con las que el nodo esta conectado, se notifica acordemente a esa billetera.
Algo similar se produce con las transacciones, si involucra a una de las cuentas se genera una notificacion que sera mostrada en la interfaz.
Cuando una de las billetera crea una transaccion, el nodo recibe una solicitud para que sea broadcasteada a sus peers, por lo cual el network listener envia una cantidad arbritaria de veces (elegimos 32) por un channel a sus workers para que quienes lean del channel envien un inv message a su peer y respondan al getdata con el tx, asi asegurando su broadcasteo.

2.3. Wallet Listener

Se tiene un hilo en el cual se espera la entrada de las diversas wallets que desean conectarse con la direccion del nodo. Cada conexion debe mandar el hash de su clave publica para poder registrarlo en el hashmap, y una vez esto este verificado y el mensaje inicial sea el correcto, se creara un nuevo thread en el cual el nodo continuara escuchando, espectante por nuevos pedidos que este pueda hacer. Si alguno de estos pedidos requiere difusion, como lo puede ser la creacion de una nueva transaccion, se pide al Network Listener que lo haga.

2.4. MerkleTree

Representado por una estructura que tiene su mismo nombre, guarda el *root* del arbol como un arreglo y las çapas"que lo completan, como un vector de vectores de *hashes*, también en formato slice. Básicamente cumple la función de un **MerkleTree**, lo construye, comprueba su root, y genera $proof of inclusions$ a través de otro struct llamado **MerkleProof**, que referencia a una prueba en particular de una transacción en particular respecto a un bloque en particular.

```rust
/// Estructura del Merkle Tree que contiene su merkleroot.
#[derive(Debug, Clone)]
3 implementations
pub struct MerkleTree {
    pub root: [u8; 32],
    hash_layers: Vec<Vec<[u8; 32]>>,
}
```

Figura 3: Estructura del Merkle Tree

3. Wallet

Cada estructura Wallet representa una billetera virtual bitcoin que ingrese el usuario al programa. Esta contiene su *publickey*, *privatekey*, un listado de sus transacciones que se *broadcastearon* a la red, el *TcpStream* por el que se conecta con el nodo y otro listado de sus $UTXO$.

```rust
#[derive(Debug)]
2 implementations
pub struct Wallet {
    pub pubkey: PublicKey,
    privkey: SecretKey,
    // transactions sent by wallet
    broadcasted_tx: HashMap<[u8; 32], BroadcastedTx>,
    node: TcpStream,
    utxos: Vec<UtxoInfo>,
}
```

Figura 4: Struct del Wallet

Esta estructura contiene todas las funcionalidades que requiere una *wallet* de nuestro software, como obtener el balance, obtener las transacciones de la misma u obtener su *address*. Al estár en comunicación con el nodo a través de su socket, el *struct* es usado como la conexión entre el programa manejado por el usuario y el nodo, por lo que tambíen incluye varias funcionalidades para pasar los *requests* del usuario (como puede ser la $proofofinclusion$ de una transacción respecto de un bloque) al nodo.

4. Protocolo Wallet Nodo

Aferrandonos al protocolo de bitcoin, decidimos crear un protocolo de mensajes que cuenta con un Header y un Payload. En el header se encuentra el nombre del request, de 12 caracteres con Null Padding, y el payload depende del pedido realizado. En el archivo de wallet_messages se cuenta con los parseos y desparseos de estos mensajes, es decir, todo lo necesario para que se pueda realizar la comunicacion entre estos mediante un stream.

5. Log File

Se cuenta con un archivo de log manejado mediante un canal, y cuyo path donde guardarlo se recibe por parametro. Todas las estructuras del nodo cuentan con una referencia a este canal, y mandan por un string formateado con lo que desean que se escriba. Desde otro thread se cuenta con el receiver del canal y el archivo abierto. Este thread escucha por el canal y ante recibo de un string se encarga de reflejar esto en el archivo, anadiendole adelante la fecha y la hora en la que se recibio dicha informacion. Lo que se guarda en este archivo son todos los mensajes que recibe el nodo una vez finalizado el block download, todos los mensajes recibidos por parte de las billeteras, y la informacion relevante de la descarga inicial de bloques.

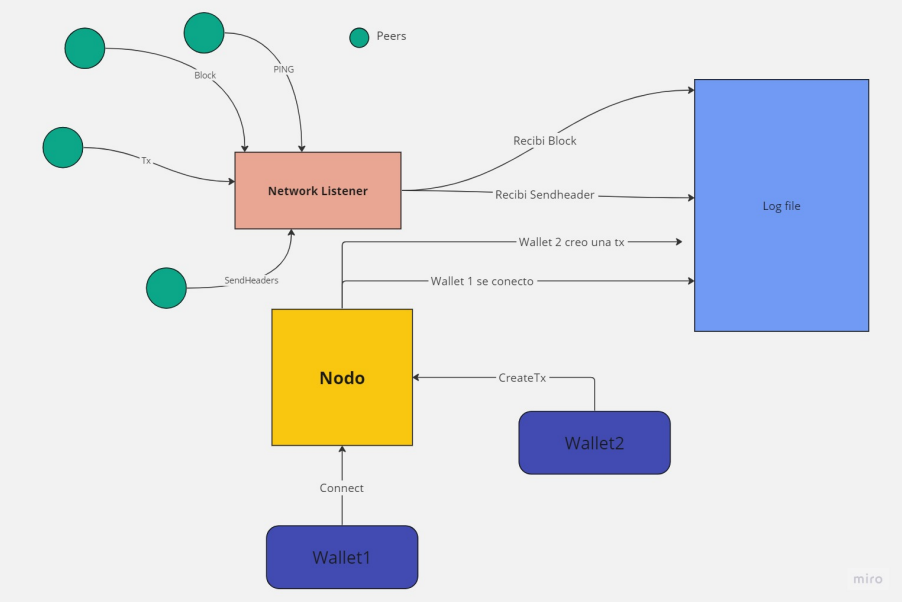


Figura 5: Ejemplo del manejo del archivo

El manejo del logfile funciona de la siguiente manera, y una vez recibe el mensaje de las diversas estructuras, el logfile hace el flush sobre el archivo.

6. Interfaz

La interfaz se implement en gtk-rs, utilizando como ayuda para el diseño, el programa Glade. Se decidió iniciar con una primera ventana que obtenga información de por lo menos una billetera para continuar con la ventana principal, la que cuenta con cuatro solapas diferentes. La primera de las solapas (Overview) cuenta con la informacion basica y general de la wallet seleccionada, el balance y las transacciones recientes. La segunda solapa, Send, se utiliza para enviar transacciones

a otras wallets por medio de su address. La tercera solapa My Transactions, informa sobre las transacciones realizadas por la wallet, esta es una lista de ID de las transacciones, que al realizar click sobre ellas se abre una ventana con la información mas detallada. La cuarta y ultima solapa contiene la información de todas las transacciones de la wallet. A su vez, cuando un error se genera, el usuario es avisado por medio de una ventana de error. Si se requiere cambiar la billetera, se cuenta con un ComboBoxText con las billeteras ingresadas previamente, y si se requiere agregar uan billetera ya existente o nueva, se cuenta con un botón para tal fin.

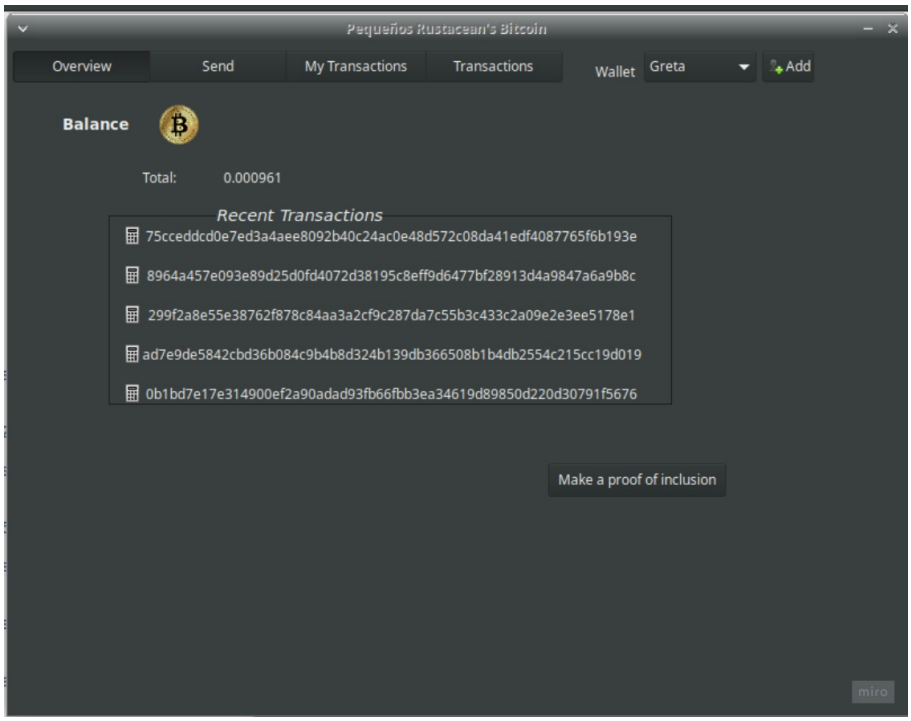


Figura 6: Pequena ilustracion de la interfaz

6.1. Gui Handler

El **GUI** (Graphic User Interface) **Handler** se encarga de manejar el estado del programa usuario. Se representa un *struct* llamado $ProgramState$, que como indica su nombre representa el estado del programa en ejecución y está conformada por un $HashMap$ de $Wallets$ accedidas a través de su nombre, una $Wallet$ seleccionada (que es la que se tiene en uso en el programa) un par de channels para comunicarse con la interfaz, y por último, una copia del $Config$ ingresado. Este se comunica con la $Interfaz$ a través de los dos channels (uno $ToGraphic$ -de la librería glib- y otro $FromGraphic$), recibiendo las entradas por interfaz, y enviandole las salidas, parseando siempre cada una de estas para, o bien mandarle a la wallet lo que necesita en su correcto formato

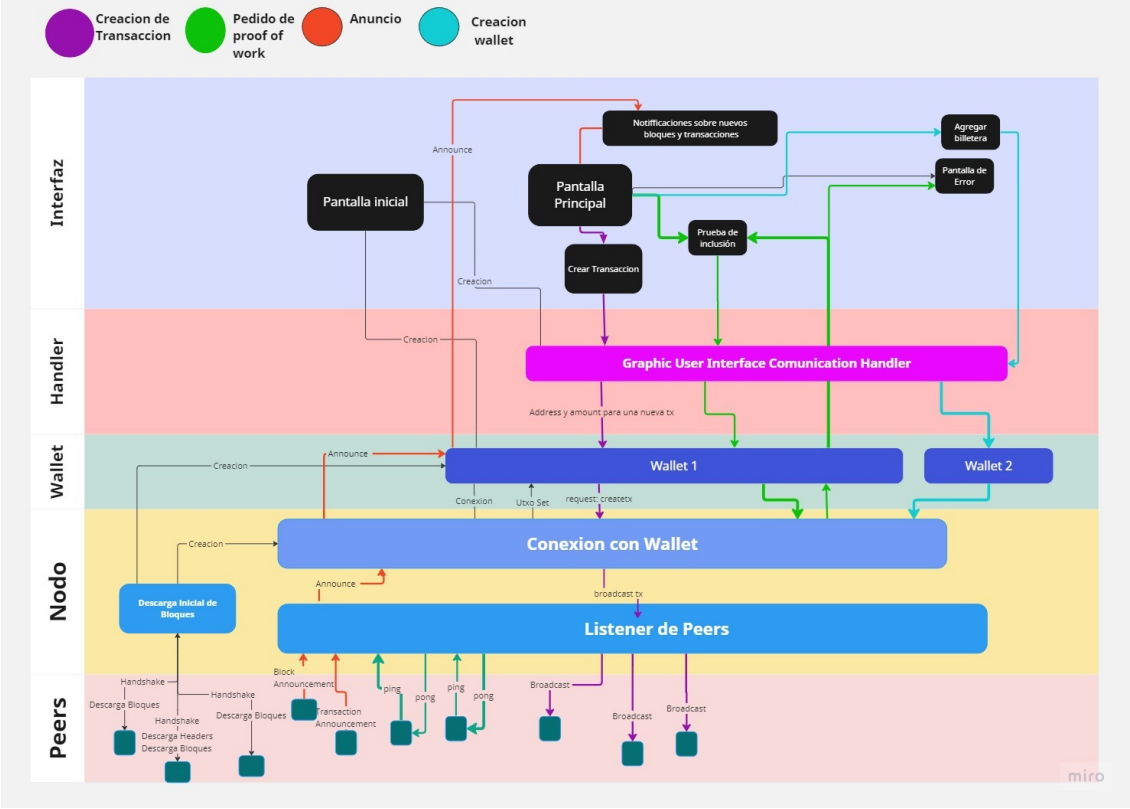
```rust
pub struct ProgramState {
    pub wallets: HashMap<String, Arc<Mutex<Wallet>>>,
    pub selected: Arc<Mutex<Wallet>>,
    pub config: Config,
    pub sender: Sender<ToGraphic>,
    pub receiver: mpsc::Receiver<FromGraphic>,
}
```

Figura 7: Struct del Program State

para llevar a cabo la funcionalidad requerida, o que la interfaz imprima por pantalla lo que se quiera comunicar al usuario.

7. Diagrama completo de interacciones

El siguiente diagrama contiene todas las interacciones que suceden durante la corrida del programa, a excepcion del archivo de log.



Como ya fue explicado mejor anteriormente, tras el click de un boton de la interfaz, se realiza un pedido a la wallet que puede o no llegar al nodo, y el nodo puede o no tener que comunicarse con sus peers, mediante su estructura.