



UNIVERSIDAD DE BUENOS AIRES

FACULTAD DE INGENIERÍA

1er CUATRIMESTRE DE 2023

Taller de Programación I

BITCOIN

Entrega final cursada

Grupo: RUSTEAM FIUBERICO

Fecha de entrega: 26/06/2023

Integrantes:

Martín Bucca	109161
Facundo De La Plata	100558
Alan Cantero	99676

Correctores del grupo:

Agustín Firmapaz
Alfonso Campodonico
Mauro Di Pietro

2. Tabla de Contenidos

Índice

1. Carátula	1
2. Tabla de contenidos	2
3. Objetivo del trabajo	2
4. Introducción	3
5. Componentes.....	3
5.1. Network	3
5.2. Handshake	3
5.3. Block	4
5.4. Initial Block Download	4
5.5. Node	5
5.6. Message Handler	5
5.7. Transacciones	7
5.8. Otros módulos	8
5.9. Utxo Set	8
5.10. Account.....	9
5.11. NodeCustomErrors	9
6. Wallet	11
6.1. Funciones de wallet	11
7. Conclusiones	17
8. Referencias	18

3. Objetivo del Trabajo

El objetivo principal del presente proyecto de desarrollo consiste en la implementación de un **Nodo Bitcoin** con funcionalidades acotadas, siguiendo las [guías de desarrollo](#) y [especificaciones](#) de Bitcoin.

El objetivo secundario del proyecto consiste en el desarrollo de un proyecto real de software de mediana envergadura aplicando buenas prácticas de desarrollo de software, incluyendo entregas y revisiones usando un sistema de control de versiones.

4. Introducción

En el presente informe, se proporcionará una descripción detallada de los componentes, módulos y arquitectura del diseño desarrollado para la implementación de un Nodo Bitcoin utilizando Rust y todas las herramientas que el lenguaje brinda. Se presentarán de manera concisa y clara los elementos clave (que, como grupo, consideramos fundamentales) que conforman la estructura del nodo, así como las interacciones y relaciones entre ellos. Esta descripción de los componentes más importantes del programa tiene como objetivo brindar una mejor comprensión de la organización y el funcionamiento global del nodo.

5. Componentes

En esta sección mostramos las distintas entidades de nuestra aplicación, debido al tamaño de los diagramas, dejamos un link a cada uno de ellos

5.1 Network

Se encarga de conectar al DNS y obtener las direcciones IP de los nodos. Se utiliza la dirección ``seed.testnet.bitcoin.sprovoost.nl`` configurada en el archivo *nodo.conf*. Implementa el Peer Discovery (punto 1.) de la conexión a la red.

5.2 Handshake

Establece la conexión con todos los nodos posible, de forma concurrente aplicando el modelo fork/join e intercambiando los mensajes versión y verack. Devuelve la lista de sockets e imprime en el log información detallada sobre cada una de estas conexiones. En caso de que alguna conexión no se pueda realizar, se imprime en el log de errores. En promedio la DNS seed suele devolver alrededor de 22 ips. Implementa la funcionalidad 2 de la conexión a la red.

5.3 Block

Representa un bloque del protocolo bitcoin. Almacena internamente todas las estructuras de datos necesarias para serializar y deserializar los mensajes.

5.4 Initial Block Download

Realiza la descarga inicial de headers y bloques (punto 3 de conexión a la red).

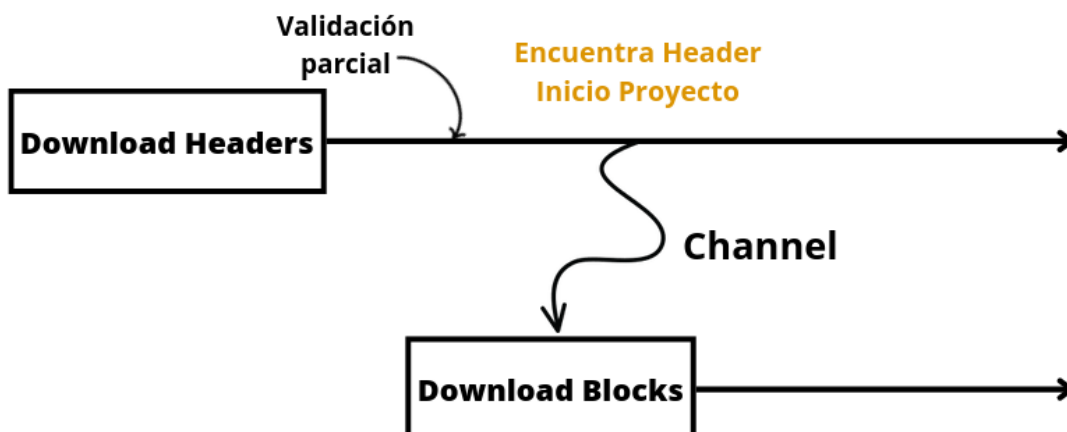
En la primera corrida, guarda en disco los primeros 2.300.000 headers para evitar descargarlos la próxima vez.

Descarga de Headers

Se ejecuta de forma secuencial, realizando peticiones a un nodo de 2000 headers cada una. Cuando se encuentra el Header que coincide con la fecha de inicio del proyecto, comienza la descarga de bloques de forma concurrente. Esto se resolvió mediante el envío de los headers al thread donde se descargan los bloques, a través de un channel.

Descarga de bloques

Aplica el modelo de concurrencia Fork/Join, descargando desde 8 nodos, cada uno en un thread. Recibe usualmente 2000 headers a través del channel y los reparte entre los 8 nodos. A su vez, cada nodo realiza peticiones de a 16 bloques. Esto permite descargar 128 bloques de manera simultánea.



Errores

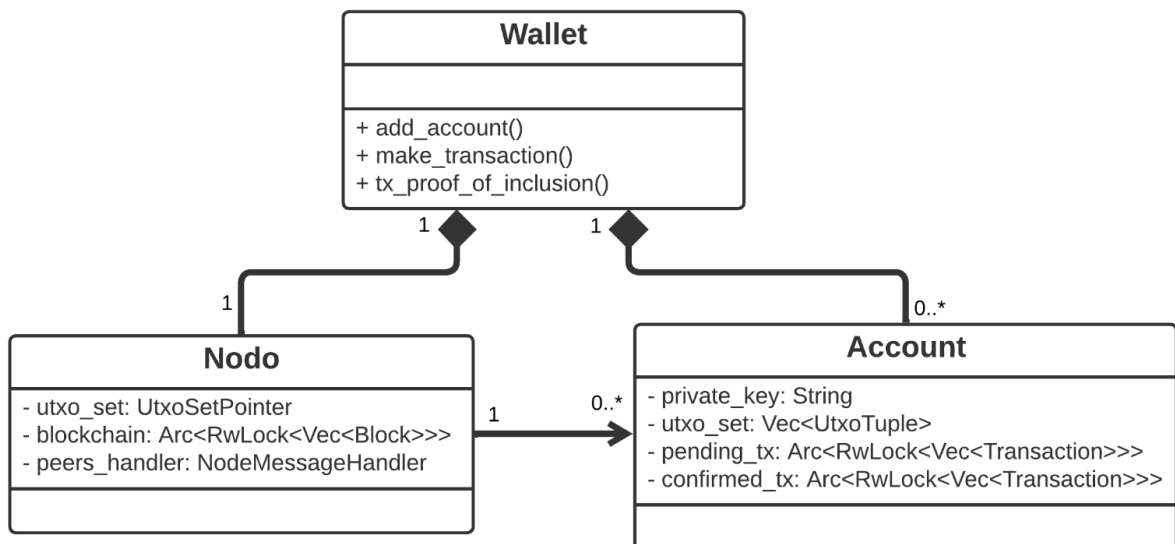
Si un nodo no responde, falla la lectura, o escritura del socket, se desconecta el nodo y el programa continúa con otro de los nodos.

Así la descarga puede continuar en caso de error.

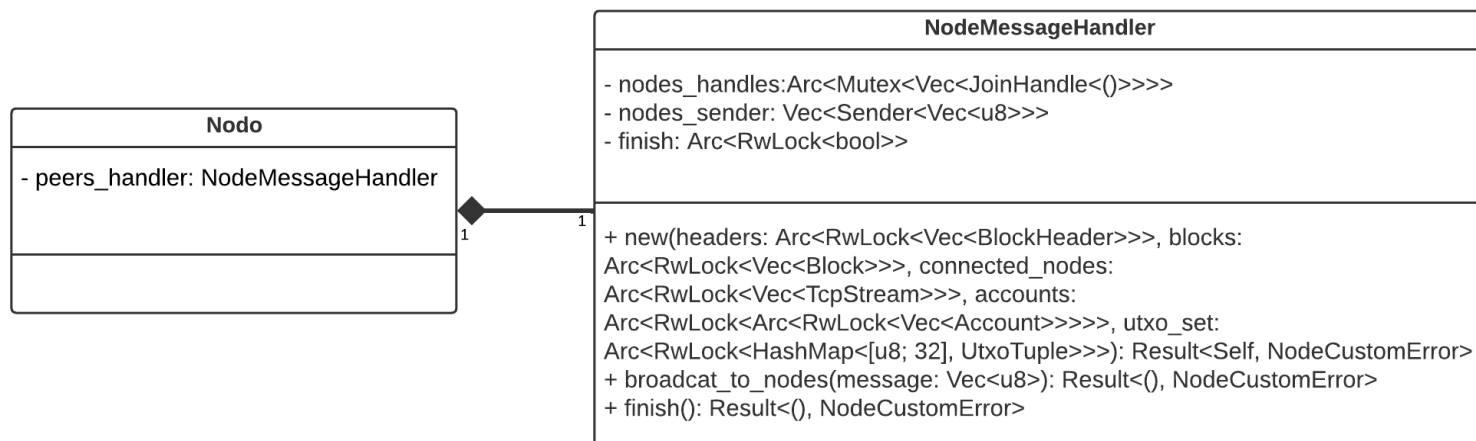
Si falla la descarga de bloques de un nodo, se devuelven los headers al channel para que se reintente desde otros nodos.

5.5 Node

Este módulo es quien almacena la blockchain, el utxo set e inicializa la escucha de nodos a través del MessageHandler. Se encarga de contestarle a la Wallet lo que le pide y también es quien se comunica con los peers conectados. Mantiene la Blockchain actualizada y el utxo set.



5.6 Message Handler



Este módulo se encarga de la comunicación entre el nodo y los peers que tiene conectados. También es donde se realiza el punto 4 de la conexión a la red: Block Broadcasting, en donde se espera por nuevos bloques para agregar a nuestra Blockchain. La estructura **NodeMessageHandler** es la encargada de esto y de manejar todos los mensajes recibidos por los peers así como los mensajes a ser enviados por nuestro nodo. El nodo debe conocer y poder comunicarle a esta estructura si quiere comenzar a escuchar, escribir algo o finalizar la conexión con el resto de los peers conectados. Es por esto que el nodo conoce e interactúa con el **NodeMessageHandler**, y este recibe y maneja las peticiones del nodo. Este Struct al ser inicializado (utilizando *new()*) inicia un hilo por cada nodo y comienza un ciclo en el cual lo primero que hace es fijarse, por el channel que recibe mensajes para escribirle al peer del thread, si tiene algo para escribir por el Tcp Stream. En caso de ser así, se ocupa de enviar el mensaje al peer. Luego de eso lee del Tcp Stream el header y payload del mensaje recibido (de contener payload) y compara el nombre del mensaje para manejarlo adecuadamente según sea necesario. Si ocurre algún error en el ciclo se imprime en el log que se desconecte el nodo y se termina el ciclo y el hilo.

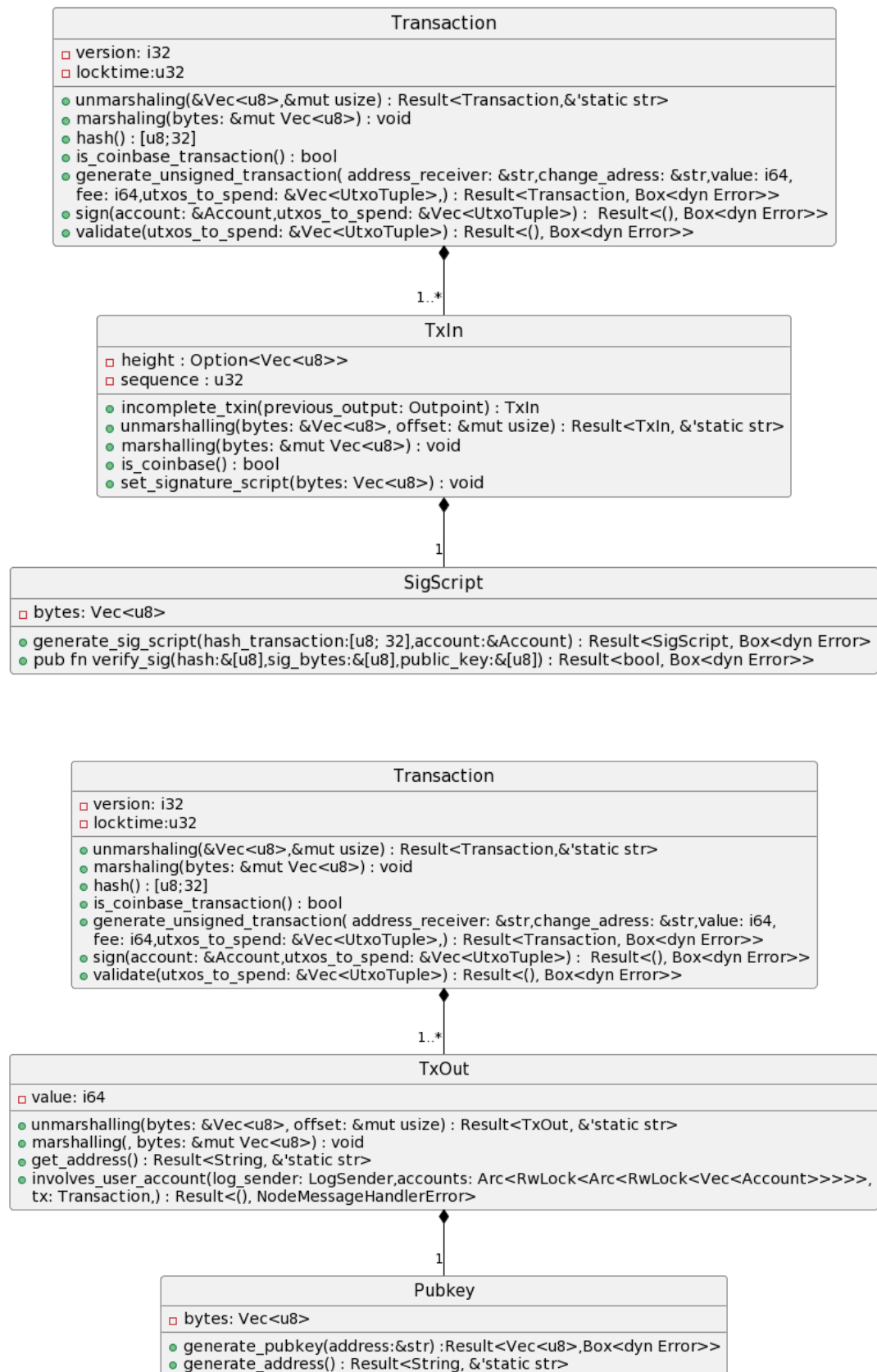
La estructura guarda un vector con los *JoinHandles* de cada hilo para que cuando se termine el programa les pueda hacer su respectivo *.join()*. También guarda un puntero a un booleano que indica si el programa debe terminarse o no. Este booleano es la condición de corte de los ciclos y en

caso de ser verdadera, se llama a *.finish()* que se encarga de cerrar correctamente los hilos.

Por último, el **NodeMessageHandler** es capaz de enviar un mismo mensaje a todos los peers que aún siguen conectados al momento de querer enviarlo. Por esta razón, decidimos que guarde un vector con los *Sender* del *channel* (cuyo par *Receiver* se encuentra en el thread de cada peer verificando constantemente si hay algo para escribir), que es creado por cada uno de los peers. En *.broadcast_to_nodes()* se recorre este vector y se envía por el *channel* el mensaje que quiere ser escrito (previamente serializado) en caso de que el peer aun siga conectado. De esta manera, logramos enviar un mensaje a la mayor cantidad de peers posibles y lograr por ejemplo, el *broadcasting* de una transacción.

5.7 Transacciones

Este módulo se encarga de crear , decodificar , codificar ,firmar y validar las transacciones . Cada transacción fue creada siguiendo la guía de desarrollo de bitcoin. A continuación se muestran los diagramas de las transacciones y sus atributos.



5.8 Otros módulos

Address Decoder

Concentra todas las funciones necesarias para decodificar las direcciones y claves utilizadas en el protocolo bitcoin.

Script

Concentra los módulos sig_script, pubkey y p2pkh_script, que son los que permiten crear y firmar los scripts para las transacciones p2pkh.

5.9 Utxo Set

Se implementó mediante un HashMap, donde la clave es el hash de la transacción y el valor es un UtxoTuple.

El UtxoTuple guarda internamente el hash de la transacción, y todas las utxos de esa transacción (TxOut e índice).

El Utxo Set se inicializa al crear el nodo y se actualiza por cada nuevo bloque que llega.

Adicionalmente, se guarda una copia de las utxos en las cuentas.

5.10 Account

Guarda la llave privada y la dirección de la cuenta.

Mantiene una copia de las utxos de la cuenta, las cuales son actualizadas por cada bloque que llega.

Se encarga de calcular su balance y realizar transacciones.

5.11 NodeCustomErrors

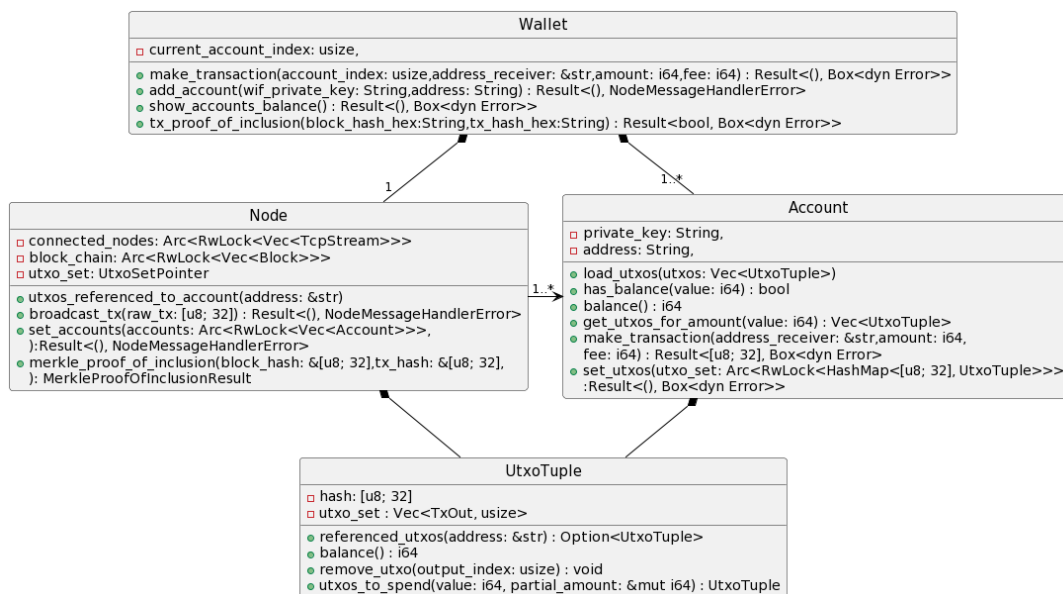
Este módulo define algunos de los distintos errores que pueden ocurrir durante el programa. Muchas de las funciones implementadas en caso de ocurrir un error, lo *mappean* al error adecuado y lo propagan hacia la función que la llama que lo maneja de igual manera hasta llegar al *main* que

lo devuelve o imprime en el log según sea necesario.

- **ThreadJoinError(String)**: Se devuelve cuando ocurre algún error al intentar hacer *.join()* a un hilo
- **LockError(String)**: Se devuelve cuando ocurre algún error al intentar obtener el *lock* de un recurso compartido entre threads
- **ReadNodeError(String)**: Se devuelve cuando ocurre un error al intentar leer de un Tcp Stream
- **WriteNodeError(String)**: Se devuelve cuando ocurre un error al intentar escribir de un Tcp Stream
- **CanNotReadError(String)**: Se devuelve cuando se intenta sacar un elemento de un vector que está vacío
- **ThreadChannelError(String)**: Se devuelve cuando ocurre un error al intentar enviar o leer algo de un *std::mpsc::channel*
- **UnmarshallingError(String)**: Se devuelve cuando ocurre un error al intentar deserializar un mensaje de bytes a *Struct <mensaje>*
- **OtherError(String)**: Se devuelve cuando ocurre un error más genérico o cuando queremos devolver un error en casos muy específicos, pero no algo puntual.

6. WALLET

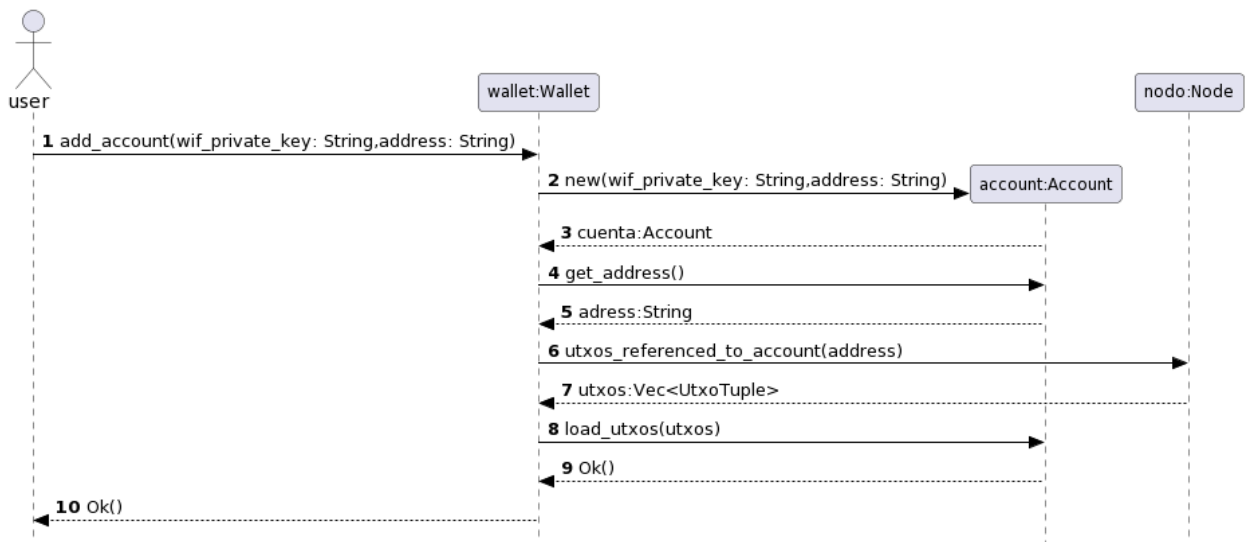
Este módulo tiene como principal objetivo la interacción del usuario con nuestra aplicación, el usuario puede ingresar una cuenta pedir el saldo de la misma o realizar una transacción . A su vez , está notificará al usuario al momento de recibir nuevas transacciones que lo involucren mostrando la respectiva información.



6.1 Funciones de Wallet

1. El usuario podrá ingresar una o más cuentas que controla, especificando la clave pública y privada de cada una:

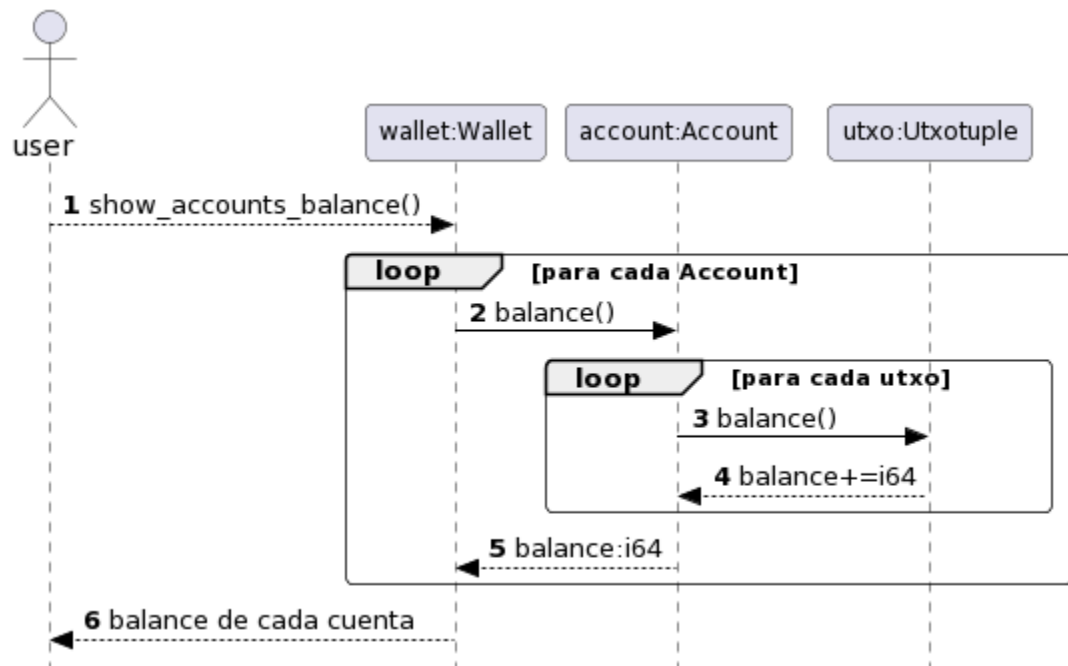
El método `add_account` de `Wallet` recibe la clave privada en formato WIF y la `address` comprimida. Se crea la cuenta, validando que los datos recibidos sean correctos (La `address` se corresponde con la clave privada) y se agrega a la lista de cuentas de la wallet.



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

Se implementa el método `show_accounts_balance`, que imprime el balance de las cuentas.

Cada cuenta se encarga de calcular su balance, recorriendo sus utxos y pidiendo a cada `UtxoTuple` su valor.



3. Cada vez que se recibe una Transacción en la red que

involucra una de las cuentas del usuario se deberá informar al usuario, aclarando que la misma aún no se encuentra incluida en un Bloque:

El `NodeMessageHandler` es el encargado de recibir mensajes y manejarlos correctamente. Cada vez que recibe un mensaje del tipo `inv`, chequea previamente dos condiciones: Que el mensaje sea avisando sobre una nueva transacción, y que la transacción que nos notifican no la hayamos recibido previamente de otro nodo (para no pedirla dos veces). En caso de que se cumplan estas dos condiciones, se le pide al nodo la transacción con el mensaje `getdata`. Cuando se recibe el mensaje `tx` se llama a la función de *Transaction*: `check_if_tx_involves_user_account(accounts)`. Esta función recorre todas las `tx_out` de la *Transaction* y por cada una de ellas llama a la función `.involves_user_account(accounts, tx)`. Se recorren todas las cuentas de la wallet y se compara la cuenta asociada a la `tx_out` y la `address` de cada cuenta. En caso de ser iguales, se notifica al usuario que una transacción que le enviaron está pendiente y falta ser agregada en un bloque. A esta cuenta se le agrega la *Transaction* al vector que guarda las transacciones pendientes, `pending_transactions`. Las transacciones que hace el usuario se agregan a este mismo vector una vez realizadas con éxito.

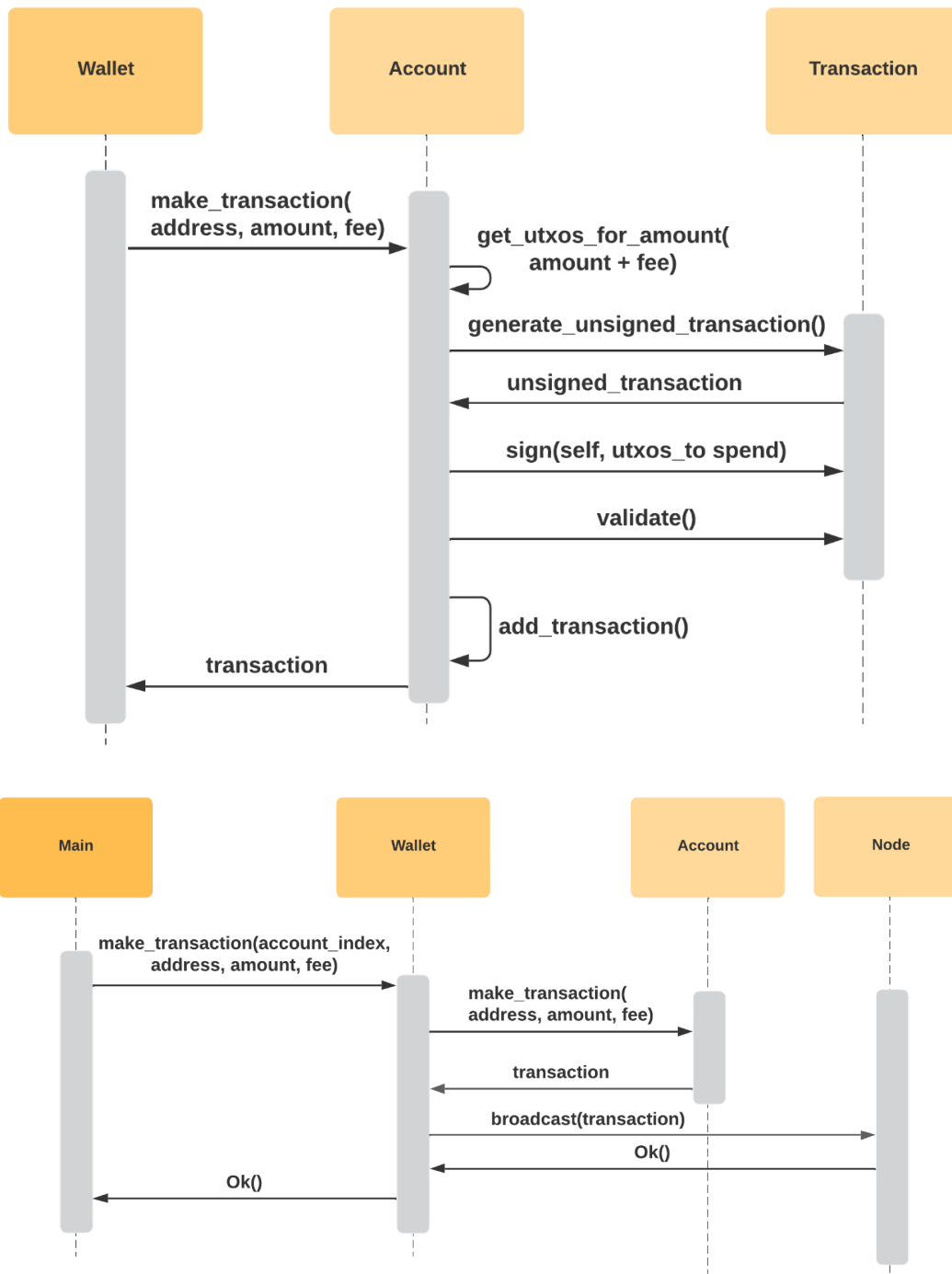
4. Cada vez que se recibe un nuevo Bloque generado se deberá verificar si alguna de las transacciones del punto anterior se encuentra en dicho bloque y se informará al usuario:

Nuevamente, el `NodeMessageHandler` es el encargado de manejar correctamente cuando llega un mensaje notificando un nuevo bloque. Luego de que el bloque sea validado y agregado a la `BlockChain` de nuestro nodo, se llama a la función `.contains_pending_transaction(accounts)`. Esta función recorre todas las transacciones del bloque y por cada una de ellas se fija si en alguna de las cuentas de la wallet se encuentra dicha transacción en el vector `pending_transactions`. En caso de que la cuenta contenga la transacción como pendiente, se le notifica al usuario que dicha

cuenta tiene esa transacción como confirmada porque llegó en ese bloque, se saca la transacción del vector *pending_transactions* y se agrega al vector *confirmed_transactions* de esa cuenta.

5. En todo momento el usuario podrá realizar una Transacción, ingresando la información necesaria para la misma. Como mínimo se deberá soportar P2PKH. La transacción generada se deberá comunicar al resto de los nodos de la red:


Aquí la wallet recibe el monto a transferir , la tarifa a pagar al minero y la dirección donde desea enviar los satoshis , con esta información la wallet pide a la cuenta (Account) que realice una nueva transacción , la cuenta busca en su propio conjunto de utxos si tiene el saldo suficiente para realizar la operación en caso afirmativo además devuelve la utxos a ser gastadas , con dichas utxos se genera la nueva transacción y la cuenta le pide a la nueva transacción que se firme y valide , terminada esta operación la cuenta devuelve la transacción para que la wallet delegue el *broadcasting* al nodo , si se realiza correctamente se devuelve un *Ok()* al usuario.

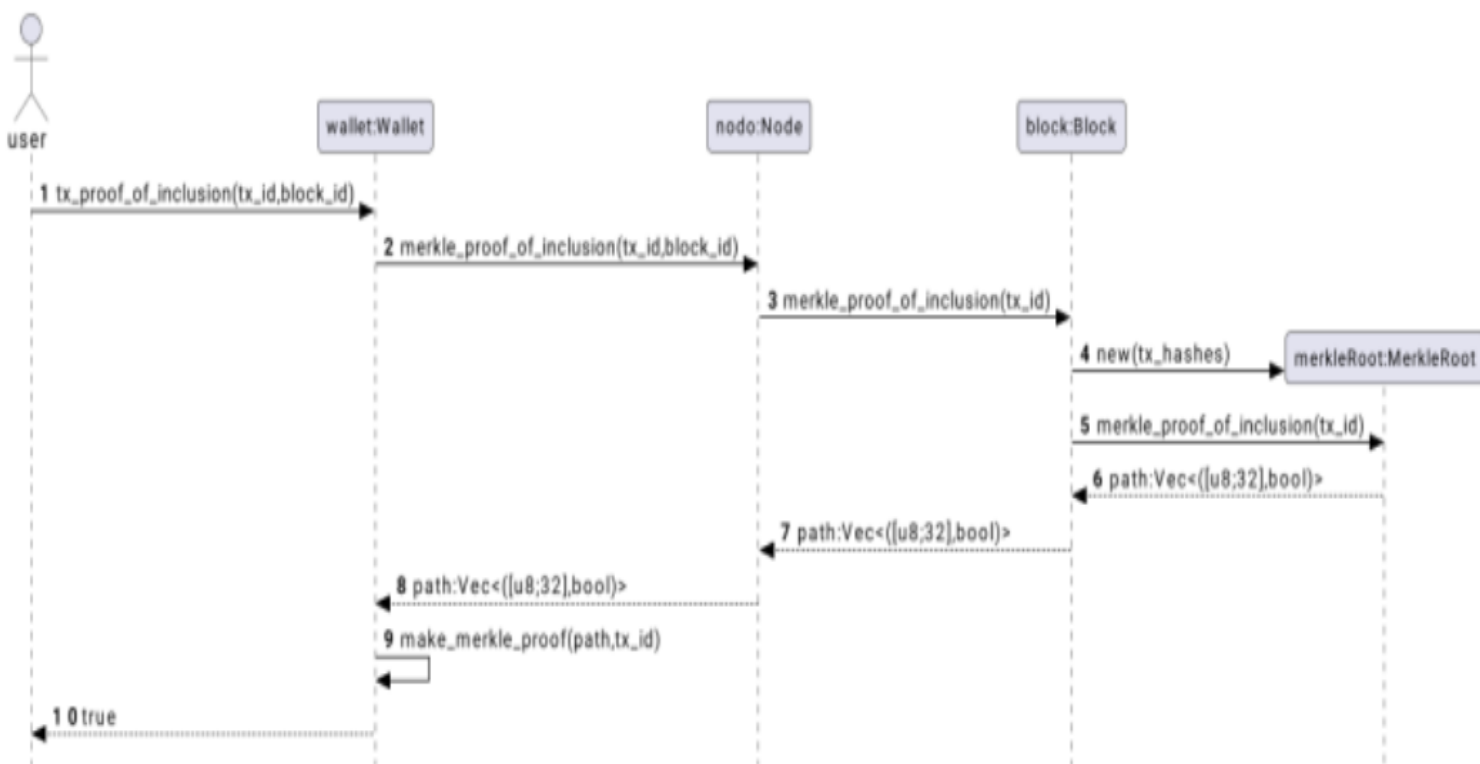


6. Adicionalmente, el usuario podrá pedir una prueba de inclusión de una transacción en un bloque, y verificarla localmente:

Una vez que se reciben los hashes de un bloque y una transacción la wallet delega al nodo la búsqueda del bloque, si lo encuentra el nodo pide al bloque la merkle proof mediante la función

merkle_proof_of_inclusion() pasándole como parámetro la transacción a buscar , el bloque generará un nuevo componente (MerkleTree) y delega a este la creación del camino de hashes que garantizan que dicha transacción se encuentra en el bloque . Luego, retornado el camino de hashes , la wallet se encarga de realizar la prueba para informar al usuario si la transacción se encuentra o no en el bloque.

Diagrama de secuencia:  merkle_proof_of_inclusion.svg



6. Conclusiones

En conclusión, este proyecto de implementación de un Nodo Bitcoin en Rust nos brindó una valiosa experiencia y aprendizajes significativos. El proyecto nos dio la oportunidad de aprender desde 0 un nuevo lenguaje de programación que nos brindó muchas herramientas interesantes para poder lograr el objetivo final del trabajo. Por otro lado, nos permitió interiorizarnos con las complejidades que implican los conceptos de Blockchain y la red de Bitcoin. No solo pudimos comprender en profundidad muchos conceptos importantes de Bitcoin, sino que también fuimos capaces de explicarlos a otras personas.

Pudimos aprender y aplicar buenas prácticas del desarrollo de Software tales como: Manejo de errores adecuadamente, Control de versiones utilizando Git y Github, planificación semanal de las distintas tareas a realizar, documentación de funciones y módulos, tests unitarios y de integración, entre muchas otras. También aprendimos conceptos fundamentales para el desarrollo del proyecto como lo fueron la programación multithreading y la comunicación vía Sockets. Sin estos conceptos hubiese sido imposible implementar el Nodo.

En adición a los aprendizajes mencionados anteriormente, también logramos adquirir la capacidad de resolución de problemas, muchos de los cuales iban apareciendo a medida que avanzamos con el proyecto. Durante estos meses, nos tuvimos que enfrentar y adaptar a nuevos desafíos que supimos resolver acordemente. El trabajo en equipo, la comunicación y coordinación eficiente fueron fundamentales para lograr el objetivo de la materia.

7. Referencias

- [Guías de desarrollo Bitcoin](#)
- [Guías especificaciones Bitcoin](#)
- [Sitio Bitcoin Developer](#)
- [Protocol Documentation Wiki](#)
- [Programming Bitcoin, Jimmy Song, O'Reilly 2019](#)
- [Mastering Bitcoin Programming the Open Blockchain, O'Reilly 2nd edition 2017](#)
- [How does bitcoin actually work?](#)
- [Bitcoin Blockchain, Miners, and Nodes \(Explained Simply\)](#)