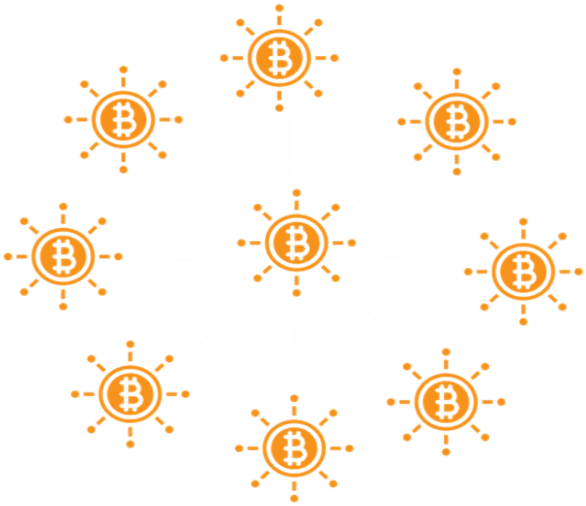
PROYECTO 1C 2023 - BITCOIN

Implementación de Nodo y Wallet Bitcoin



Brandon Schiffer, Valentina Adelsflügel, Iñaki Llorens, Carolina Mauro

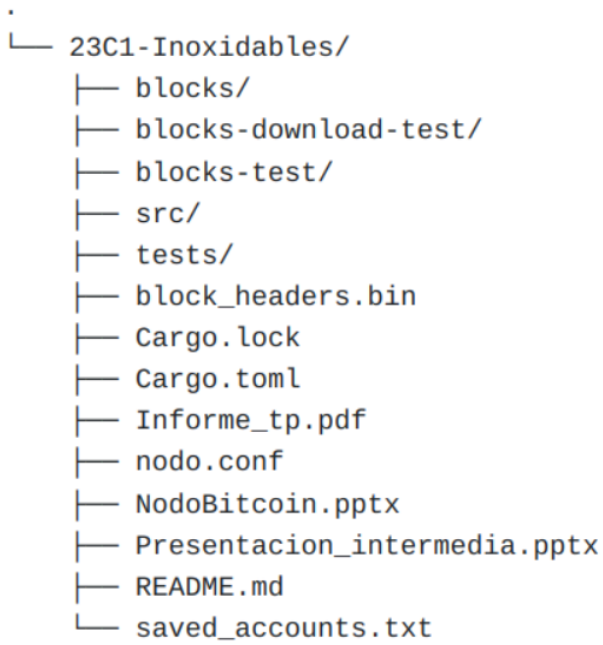
26/06/2023

ÍNDICE

Introducción

En el presente informe se detalla a modo explicativo las ideas para llevar a cabo este trabajo, la arquitectura del diseño desarrollado para este proyecto y las operaciones más relevantes. Cada sección corresponde a uno de los puntos a desarrollar y es acompañada por diagramas que ilustran las explicaciones. Por último, se encuentra la conclusión de este proyecto, que encapsula todo lo llevado a cabo durante el cuatrimestre para la gestión y realización del proyecto.

Componentes y módulos de la arquitectura general del diseño del proyecto



En un primer nivel, el proyecto se encuentra dividido en: el archivo de configuración (“nodo.conf”), la carpeta de tests, la carpeta src, la carpeta blocks-test y la carpeta blocks.

En el archivo de configuración, se setea la DNS correspondiente, el puerto utilizado, la versión de Bitcoin que decidimos usar (70013), la IP local, el TimeStamp correspondiente a la fecha de inicio del proyecto (10 de abril), el path del log y el path de la carpeta blocks.

En cuanto a la carpeta tests, contiene el archivo con los tests de integración, donde se prueba la funcionalidad completa de la descarga de bloques, las funcionalidades de la Wallet y la conexión entre el Nodo y la Wallet.

La carpeta blocks-test contiene ciertos bloques descargados que son utilizados para el test de integración correspondiente.

En la carpeta blocks, se almacenan todos los bloques descargados, tanto en el Initial Block Download como durante el Broadcasting.

En la carpeta de src, se encuentran todos los módulos y componentes restantes correspondientes al desarrollo de este proyecto. Iremos desglosando cada módulo y explicando sus componentes. Cada módulo tiene una razón de ser y un propósito, y sus componentes se vinculan directamente.

En primer lugar, decidimos tener un módulo “block”, que contiene lo correspondiente al BlockHash, Merkle Tree y Proof of Inclusion. Aquí se desarrollan los temas relacionados con la validación del bloque: el merkle root, la proof of work, proof of inclusion, etc.

En segundo lugar, tenemos un módulo “block header”, donde su principal componente representa al block header. Aquí, se desarrolla su estructura junto a su implementación, es decir, sus funcionalidades, como ser un constructor, la serialización y deserialización, etc.

En tercer lugar, decidimos tener un módulo para los canales, donde se encuentran las componentes del canal para la UI y el canal para la Wallet. Ambos poseen un sender y receiver e implementan sus métodos para mandar y/o recibir información entre diferentes threads.

En cuarto lugar, la serialización en Bitcoin maneja mucho el concepto de Compact Size, por lo que decidimos hacer un módulo específico para ello, donde se representa al Compact Size como un Enum con su implementación para leer “variable integer” y

retornarlo como un CompactSize, convertir de Compact Size a byte, etc.

En quinto lugar, tenemos el módulo de “config”, donde se resuelve todo lo relacionado con el archivo de configuración. Aquí, se cargan los datos del archivo y se setean las variables de entorno correspondientes.

En sexto lugar, decidimos tener un módulo para los “connectors”. En esta sección, están los componentes del dns_connector y el peer_connector. El dns_connector es una estructura que contiene a la “dns hostname” y el número de puerto y es usado para conectarse al servidor DNS correspondiente. El peer_connector contiene las funciones para mandar y recibir mensajes por el Stream.

En séptimo lugar, tenemos el módulo del “header”. Aquí, se implementa la estructura del header, con sus campos correspondientes y sus funcionalidades, incluyendo un constructor, la deserialización, extracción del nombre del comando, etc.

En octavo lugar, se encuentra el módulo del logger. Aquí, se maneja lo relacionado con el Archivo de Log, donde se mantiene un registro de los mensajes recibidos por el Nodo. El Logger es el encargado de escribir los mensajes recibidos por el Nodo en el archivo de log.

Luego, decidimos tener un módulo que contiene cada uno de los mensajes que manejamos del protocolo de Bitcoin. Es decir, cada componente del módulo “messages” es un mensaje correspondiente al protocolo Bitcoin. Los mensajes que necesitamos implementar aquí fueron: Block Message, GetData Message, GetHeaders Message, Headers Message, Inv Message, Tx Message, Version Message y Verack Message. Los mensajes de Version y Verack fueron utilizados para el Handshake, el Inv y el GetData para el broadcasting de bloques y transacciones, el Tx para recibir y mandar transacciones, el Headers y el GetHeaders para lo relacionado con la descarga de Headers Inicial y el Block Message para descargar los bloques tanto en el IBD como durante el broadcasting.

A continuación, tenemos un módulo específico para el nodo. Este módulo contiene diferentes componentes: block_header_downloader, message_type, mod (archivo principal), read y receive_messages. El BlockHeaderDownloader es una estructura que maneja la descarga inicial de Headers. En el message_type decidimos implementar un enum para los diferentes tipos de mensaje que debe manejar el nodo (los correspondientes al módulo “messages”). En el mod, se encuentran las funciones principales del nodo, como ser la función de handshake, la inicialización de la conexión con un peer, la descarga de headers y bloques, el manejo de broadcasting. En la

componente read, se encuentra todo lo referido a la lectura del path del bloque (cada bloque es guardado con el nombre de su hash, es decir, “hash del bloque.bin”), los block headers y el timestamp, entre otros. Por último, en receive_messages se encuentra la lógica de los mensajes recibidos y mandados por el nodo. Aquí, se maneja el tema de mandar un Pong como respuesta a cada Ping de los peers, el manejo de la recepción de mensajes como addr, feefilter, inv, tx, etc. Además, se maneja la recepción del envío de las cuentas de usuario por parte de la Wallet.

Asimismo, poseemos un módulo “node-error”, donde su principal componente es un enum de todos los mensajes de error que pueden ocurrir durante la ejecución del programa. Nos pareció buena idea tener un enum para los diferentes mensajes para poder hacer un buen uso y administración de los casos de error.

Por otro lado, decidimos tener un módulo para lo referido a la implementación de los “pools”. Para la gestión de diferentes hilos (uno para cada peer conectado), nos pareció buena idea implementar un ThreadPool. De esta forma, podíamos gestionar eficientemente el uso de varios threads. En este módulo, están los siguientes componentes: block_downloader, block_downloader_pool, listener, message_listener_pool y received_data_listener. El BlockDownloaderPool es una implementación particular de un ThreadPool, con el objetivo de gestionar la descarga de bloques en paralelo por los diferentes hilos. Su estructura posee un vector de BlockDownloaders, que son una estructura que representa un hilo trabajador en la ThreadPool. En este caso, su propósito es la descarga de bloques y guardarlos en la carpeta blocks. El MessageListenerPool es otra implementación de una ThreadPool, pero en este caso, su propósito es gestionar a los hilos trabajadores relacionados con escuchar mensajes que broadcastean los otros nodos. Entonces, el MessageListener es una estructura que también representa un hilo trabajador en una ThreadPool, pero en este caso, con otro propósito. Aquí, el MessageListener estará encargado de la escucha de mensajes durante el Broadcasting. Para diferenciar entre lo que es Broadcasting de transacciones y bloques, creamos un enum RecievedDataFromPeers, el cual diferencia entre un BlockHash o una Transaction.

Por otra parte, decidimos crear un módulo específico para las transacciones. Sus componentes son: outpoint, tx_input, tx_output, utxo_set y transaction. Las primeras cuatro componentes corresponden a los elementos que componen a una transacción y la última componente utiliza cada uno de los elementos anteriores para su implementación. Es decir, existe una relación directa de dependencia en este módulo entre sus componentes y la transacción.

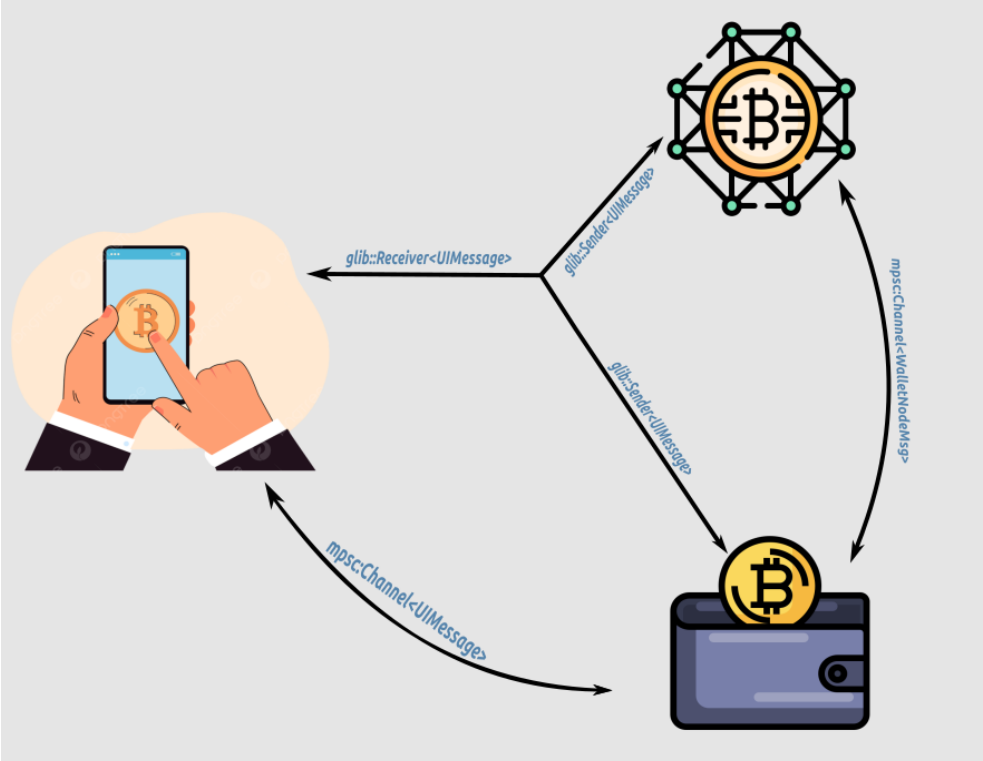
Además, nos pareció organizado tener un módulo para lo que respecta a la UI. En este módulo, tenemos los assets, components, css, utils, el builder, ui message y el window.glade. En assets decidimos incluir lo referido a los íconos. En components, tenemos los principales componentes de la interfaz gráfica como la main_window, login_window y las diferentes páginas de la interfaz. En CSS, tenemos el style utilizado. En Utils, decidimos tener un archivo con diferentes funcionalidades que nos podían resultar útiles para la interfaz gráfica, como la creación de labels o crear la información para el bloque descargado. En builder, tenemos lo referido a cargar el CSS, y correr la interfaz gráfica. Asimismo, decidimos implementar un enum para los diferentes mensajes que puede tener la UI. De esta forma, podíamos manejar más eficientemente lo referido a los BlockHeaders, bloques y transacciones por separado. Por último, en glade está lo que fuimos construyendo utilizando esta herramienta.

También, decidimos tener un módulo de Utils fuera de la UI, para utilidades generales. Aquí tenemos funciones que pueden ser utilizadas en diversos módulos, como por ejemplo chequear si un TCP Stream está conectado.

Finalmente, tenemos un módulo específico para la Wallet. Aquí dentro, se desarrolló todo lo referido a la segunda parte del proyecto. Sus componentes son: account, bitcoin_address, node_wallet_message, wallet_account_info y wallet_impl. La estructura Account cuenta con su Bitcoin Address asociada, la private key, el utxo_set, el balance, las unconfirmed transactions y las confirmed transactions. El Bitcoin Address es una estructura que posee la address correspondiente e implementa lo referido al pk_hash y pk_script. Además, decidimos tener 1 enum para mensajes de la wallet al nodo y viceversa. De esta forma, podemos manejar eficientemente el envío de información entre el nodo y la wallet. wallet_acount_info representa una estructura que guarda información de la wallet. En wallet_impl, está lo referido a la implementación de la wallet, donde su estructura contiene todas sus cuentas asociadas.

Cabe destacar que también tenemos un archivo donde definimos todas las constantes que se usan en el proyecto y un archivo main que corre la UI y por ende luego todo lo referido al nodo y la wallet.

La conexión entre los 3 grandes módulos de nuestro proyecto se da de la siguiente manera:



La UI se comunica con la wallet y el nodo, mientras que entre el nodo y la wallet se envían información que necesiten respectivamente mediante un channel. Por ejemplo, para el chequeo de si una transacción entrante a la red se encuentra ligada a una de las cuentas del usuario, el nodo le pide a través del canal a la wallet las addresses, éste se las manda y así el nodo puede verificar por cada tx recibida si está ligada a una de las cuentas.

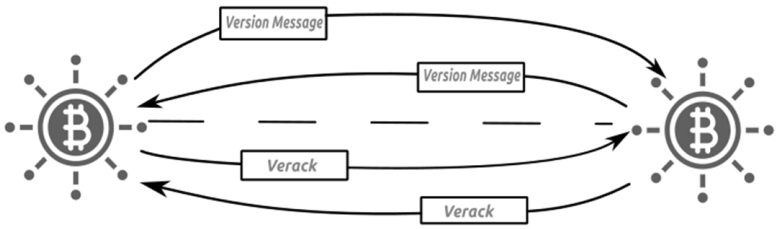
Secuencia de las operaciones más relevantes

Vamos a diferenciar dos grandes aspectos para las operaciones más relevantes de este proyecto: lo referido al nodo y aquello correspondiente a la wallet.

En cuanto al nodo, la secuencia de las operaciones más relevantes es:

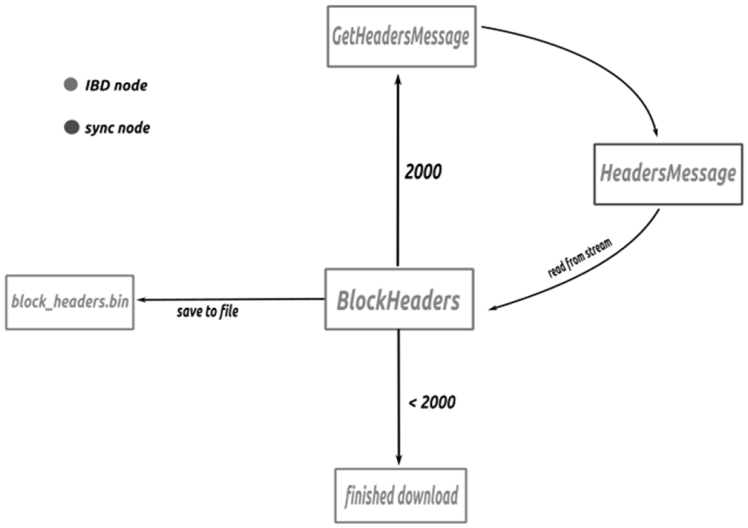
En primer lugar, es necesario conectarse a la red de Testnet para poder realizar la implementación de nuestro nodo Bitcoin. Para ello, el primer paso es realizar el Peer Discovery. Mediante el DnsConnector mencionado anteriormente, nos conectamos al server DNS correspondiente y de esta manera obtenemos las ips. Estas ips serán los peers a los que estaremos conectados para el manejo de bloques y transacciones.

Una vez obtenidas las ips, el nodo procede hacer el Handshake. En un principio, se hace el Handshake únicamente con un Peer para la descarga inicial de Headers, y luego se realiza el Handshake con los restantes Peers para la descarga de bloques en paralelo. En este proceso de Handshake, la secuencia es: mandamos nuestro Version Message, recibimos el Version Message del nodo al cual nos estamos conectando, le mandamos nuestro Verack Message en confirmación a la recepción de su Version Message y recibimos su Verack Message en confirmación a la recepción de nuestro Version Message.

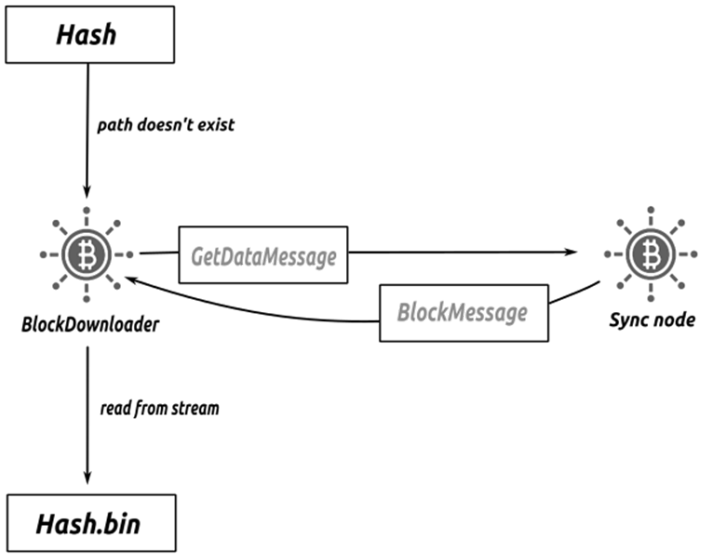


Luego, se procede a realizar la descarga inicial de Headers con nuestro Peer principal. Para ello, le mandamos el mensaje GetHeaders, recibimos su mensaje de Headers y

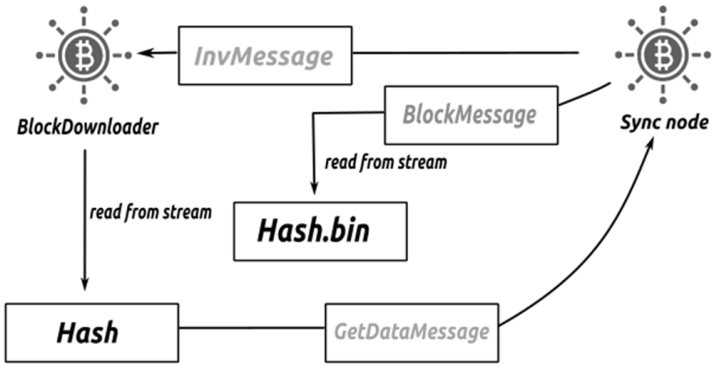
descargamos los blockHeaders. Si nos llegan 2000, volvemos a mandar el GetHeaders y si llegan menos de 2000, significa que terminó la descarga. Así, obtenemos inicialmente la descarga de todos los blockHeaders. Además, durante esta descarga el nodo nos puede enviar diferentes mensajes como el Ping, por lo tanto aquí también manejamos esos mensajes respondiéndole correspondientemente.



Durante la descarga inicial de bloques, aquí ya entran en juego todos los Peers. Para llevar a cabo esta descarga, se decidió hacerlo en paralelo para obtener mayor eficiencia y rapidez. Para ello, se implementó el BlockDownloaderPool anteriormente mencionado, el cual gestiona los hilos para la descarga de bloques en paralelo. Cada hilo representa a un Peer conectado. Para esta descarga de bloques, en el hilo principal se encolan los hashes de los blockHeaders, y así la pool gestiona cada hash para un hilo entonces cada Peer va a descargar un bloque diferente. Para la descarga, cada BlockDownloader (que representa un hilo) manda el GetData Message al nodo conectado, y éste nos envía un BlockMessage con el bloque y es ahí donde se descarga en la carpeta blocks.



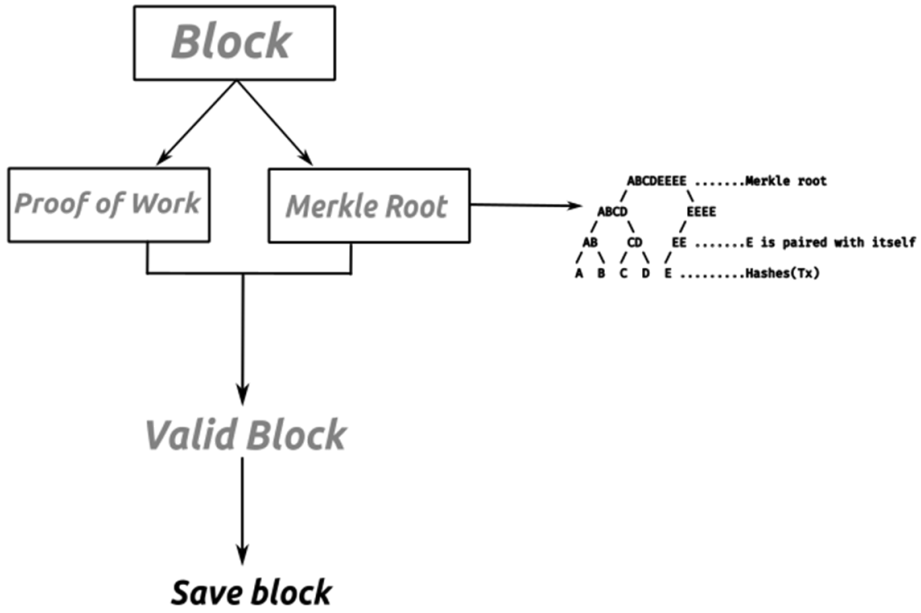
Posteriormente, se procede al Broadcasting. Aquí, el proceso es el siguiente: recibimos un InvMessage del nodo conectado, el MessageListener obtiene el hash del bloque de ese Inv, y con ese hash manda el GetData correspondiente. El nodo nos mandará el BlockMessage con el bloque y así lo descargamos de forma similar al IBD. Cabe destacar que para este proceso se implementó otra Pool, la MessageListenerPool, para manejar el Broadcasting por separado al IBD.



Antes de guardar el bloque, se lo debe validar. Para ello, se lleva a cabo los procesos del Merkle Root y la Proof of Work. Una vez que el bloque pasa estas pruebas, sabemos que éste es válido y entonces sí se guarda en nuestra carpeta de blocks.

Además, durante el IBD se actualiza el utxo_set por cada bloque recibido y también por

cada bloque ya en el Broadcasting.



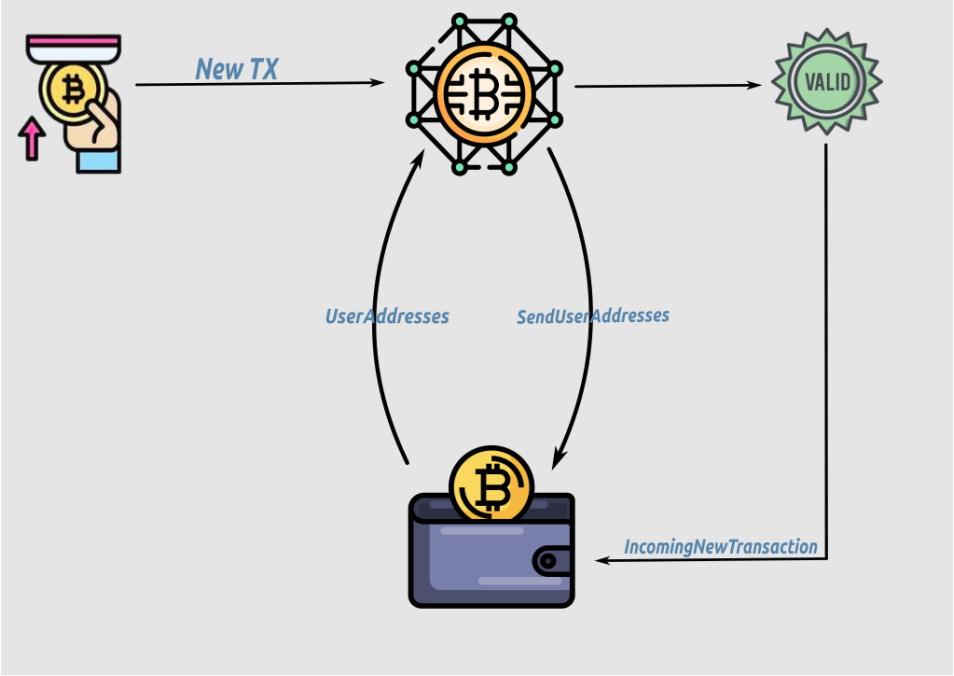
En cuanto a la Wallet, la secuencia de las operaciones más relevantes es:

En primer lugar, se monitorea por Outputs. Posteriormente, se crean transacciones sin firmar. Se firma la transacción y se broadcastea.

Para ello, fue necesario implementar varias funcionalidades. Primero, implementamos la abstracción de una cuenta, es decir, que un usuario pueda ingresar una o más cuentas que controla mediante la clave pública y privada de cada una. Por lo tanto, la wallet creada tendrá las cuentas del usuario. Segundo, se implementó que para cada cuenta se pueda visualizar el balance de la misma, o sea, la suma de todas sus UTXO disponibles al momento. En este proceso, realizamos lo siguiente: recorremos el utxo_set y por cada tx_output en tx_outputs conseguimos el pk_hash. A continuación, si este pk_hash es igual al del usuario, se suma al balance el valor de la tx_output y así sucesivamente hasta conseguir el balance final.

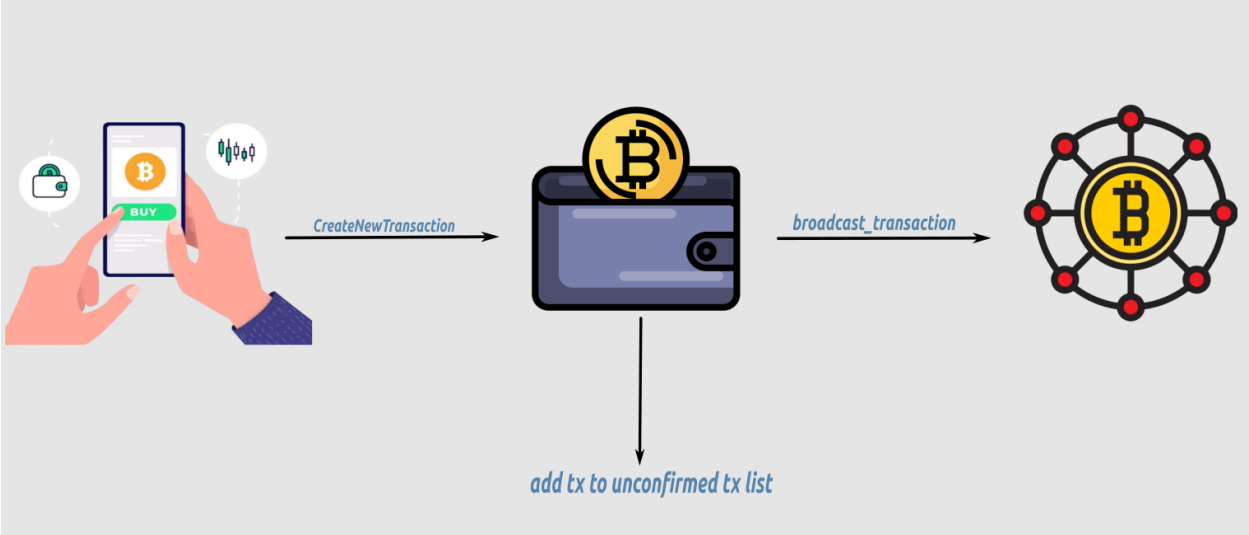
Luego, implementamos la funcionalidad de que cada vez que se recibe una transacción en la red que involucra a una de las cuentas del usuario, se le informe a éste, avisándole que aún no se encuentra en un bloque. Para ello, durante el Broadcasting, el MessageListener se queda escuchando por Inv Messages de tipo MSG_TX. Si es así, le

enviamos al nodo un GetData correspondiente y recibimos un Tx Message. Al recibir este mensaje, leemos la transacción y el nodo le manda a la Wallet que le envíe las cuentas de usuario que tiene. El nodo recibe estas cuentas de usuario, y verifica si la transacción recibida está ligada a alguna de las cuentas del usuario. Si es así, el nodo le manda una señal a la Wallet indicando que la transacción recibida involucra a una de sus cuentas, especificando qué cuenta es.



Asimismo, cada vez que se recibe un bloque nuevo, se verifica si alguna de las transacciones del punto anterior se encuentra en dicho bloque y se le informa al usuario. Para ello, cada transacción ligada a una de las cuentas del usuario es guardada como transacción pendiente. Al llegar un nuevo bloque, se recorren todas las transacciones para verificar si hay alguna que tenga que ver con las cuentas del usuario. Para esto, utilizamos el utxo_update. Además, se le manda un mensaje a la Wallet indicando que llegó un nuevo bloque y se actualiza el utxo_set. Si en ese bloque hay alguna de las transacciones que teníamos pendientes, se saca de aquí y se pone como confirmadas.

Por otro lado, el usuario tambien puede crear una transacción. La secuencia es aquí primero crear las transacción sin firmar y luego firmarla. Una vez completado este proceso, la transacción ya está lista para ser broadcasteada. Para ello, la transacción es mandada al nodo y es éste quien se encarga de comunicar al resto de nodos en la red.



Finalmente, se implementó la proof of inclusion. El usuario aquí puede pedir una prueba de inclusión de una transacción en un bloque y así poder ser verificada.

Nodo como servidor

Para la entrega final del trabajo se nos asignó la tarea de implementar una funcionalidad extra para el nodo, esta es la de que nuestro nodo bitcoin sea capaz de actuar de forma de servidor. Esto implica que ahora el nodo va a poder actuar de la misma forma que los demás peers de la red, permitiendo que un nodo cliente se conecte y realice la descarga inicial de headers y bloques.

Para poder probar el funcionamiento del nodo como servidor necesitamos tener dos instancias de nuestro programa corriendo, una que actúa como servidor y una que se conecta a nuestro servidor, a esta la llamamos cliente. Para poder hacer esto decidimos que se debe pasar el archivo de configuración .conf como argumento del programa, si no se pasa nada por default se usará el archivo nodo.conf. En este archivo se especifica la lista de ips y puertos de los nodos peers a las que se quiere conectar, además de las que se reciben desde la DNS de testnet. En el archivo .conf del cliente agregamos acá la ip y el puerto de nuestro nodo servidor para que pueda conectarse. Además en el archivo de configuración se debe especificar el path de la carpeta donde guardar los bloques y el path al archivo .bin donde se guardan los headers.

Para crear la funcionalidad de servidor se crea un thread nuevo luego de la descarga inicial de bloques, este ejecuta la función start_server(), esta función crea un TcpListener con la IP y el puerto de nuestro nodo servidor. Usando un for, que crea una conexión por medio de un stream usando la función listener.incoming(), creamos un thread por cada conexión que le llegue al servidor. Estos threads ejecutan una función que se ocupa de manejar los mensajes que el servidor va recibiendo por medio del stream. Primero se realiza un handshake con el peer que se está intentando conectar al servidor y si este sale bien, es decir recibimos correctamente el version y el verack message, se continúa a manejar los diferentes mensajes que nos puede mandar el cliente. Nuestro servidor puede responder a los mensajes GetHeaders y GetData, estos corresponden a la descarga de headers y bloques.

Para responder al mensaje GetHeaders se envía al cliente el mensaje Headers. El mensaje que recibimos del cliente nos indica a partir de qué block header debemos empezar a mandarle headers. Lo que implementamos en un principio del lado del servidor fue que por cada mensaje GetHeaders que se recibía, se abría el archivo que guarda los headers y buscaba el header pedido. Esto funciona en un principio pero se va haciendo muy costoso de tiempo y recursos a medida que la descarga de headers va progresando, porque cada vez hay que buscar en el archivo por el header que nos mandan, y cada vez este header está en una posición más adelante en el archivo. Como optimización se nos ocurrió que podríamos mantener el archivo de headers abierto, esto por un lado hace que no tengamos que abrir y cerrar constantemente el archivo, y por otro lado guarda la posición donde estaba el último header que mandamos, entonces cuando nos llegue un nuevo mensaje de GetHeaders tarda O(1) en encontrar el nuevo header, ya que este siempre va a ser el próximo que quedó sin mandar. Por otra parte el mensaje Headers manda los 2000 headers siguientes al header que se recibe en GetHeaders, si es que los hay, sino manda todos los restantes.

Responder al mensaje GetData es un poco más simple ya que es bastante directo lo que hay que enviar de vuelta. En la descarga inicial de bloques, el mensaje GetData pide bloques en base a un hash del bloque. La respuesta a este mensaje es el mensaje Block y lo que hace el servidor para enviarlo es buscar en la carpeta donde se guardan los bloques el que fue pedido y enviarlo en forma de bytes por medio del stream.

Conclusión

Como conclusión, queremos destacar que cada uno de los puntos del alcance de este proyecto fueron desarrollados en grupo, discutiendo las decisiones tanto de implementación como de arquitectura del diseño en conjunto. Logramos llevar a cabo eficientemente un proyecto real relacionado a Bitcoin, tanto el nodo como la wallet, intentando aplicar todas las buenas prácticas vistas durante el cuatrimestre. Además, intentamos hacer uso lo más idiomático posible del lenguaje Rust. Cada requerimiento funcional fue pensado detalladamente, hasta llegar a lo que creemos como la mejor solución para resolver la funcionalidad. El trabajo en equipo nos permitió dividir tareas y hacer un mejor uso del tiempo para poder llevar a cabo y cumplir con los requisitos de este proyecto.

Referencias

- **Sitio Bitcoin Developer:** https://developer.bitcoin.org
- **Protocol Documentation Wiki**: https://en.bitcoin.it/wiki/Protocol_documentation
- **Programming Bitcoin**, Jimmy Song, O'Reilly 2019: https://www.oreilly.com/library/view/programming-bitcoin/9781492031482/