Proyecto: Bitcoin - 1er Cuatrimestre 2023

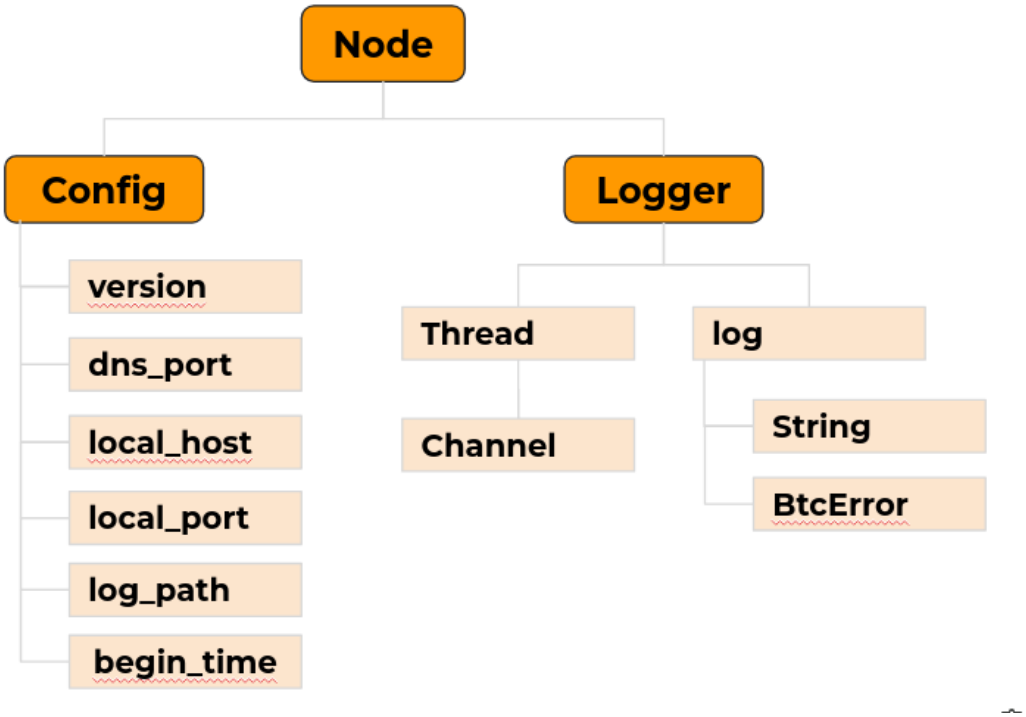
Catedra Deymonnaz

Papas Rusticas

Integrantes	
Nombre	**Padrón**
Aschieri, Juan Pablo	108000
Natalini, Marco	108056
Rovira Rossel, Francisco	107536
Stiefkens, Julián Melmer	108111
Corrector:	Miletta, Martín

Este informe tiene la finalidad de explicar las principales estructuras del Trabajo Práctico que decidimos implementar, adjuntando diagramas para facilitar su entendimiento.

Estructuras Config y Log



Primero, antes de empezar con la implementación del nodo en si, hablaremos de unas estructuras que nos facilitan la instanciación y el seguimiento de las acciones del mismo. Estas estructuras son **Config** y **Logger**.

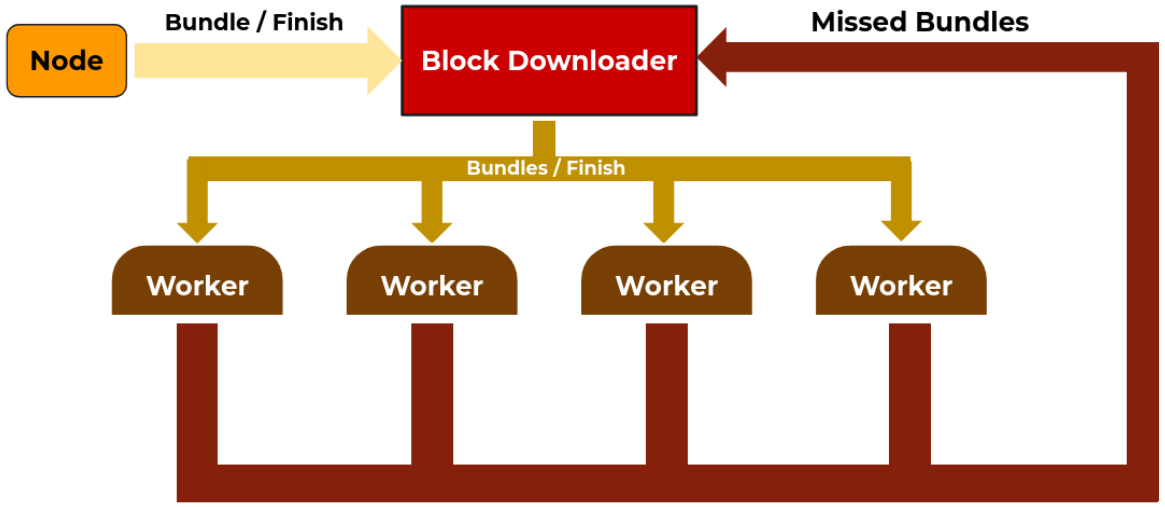
La estructura **Config** es utilizada para poder pasarle un archivo de configuración al nodo (llamado *nodo.conf*), el archivo es manejado al principio del programa cuando el nodo es instanciado porque requiere la inicialización de sus respectivos campos con aquello que haya en el archivo de configuración pasado por parametro en el inicio del programa.

Por otro lado, el **Logger** se encarga de escribir en un archivo llamado *node_log.txt* todos los mensajes recibidos por el nodo, asi como los eventos que este realiza (como iniciar el BlockDownloader, iniciar la carga de datos a memoria y a disco, la creación del set de UTXO, y errores que ocurren en la ejecución). El log sirve mucho

a la hora de ver realmente lo que está pasando con el nodo, y lo usamos a la hora de cargar la aplicación para que el usuario pueda ver el progreso de inicialización del Nodo. Usa un thread aparte para ir escribiendo en el archivo asi no interfiere con la ejecución del programa, y se comunica a través de un channel entre el nodo y el logger.

Block Downloader

Para tener un modelo de descarga de bloques en paralelo, la estructura que diseñamos para esto lleva el nombre de **BlockDownloader**



Cada Worker se ejecuta en un thread aparte, cada uno conectado a un peer distinto, para pedirle un Bundle de bloques. Definimos un bundle como un Vector de 16 bloques para descargar. El BlockDownloader se comunica con los workers mediante un channel compartido en donde le llegan del nodo los Bundles a descargar, el worker que adquiera el lock del channel es el que le pide a su respectivo peer los bloques que obtuvo del channel, de esta forma si se mandan muchos bloques, estos pueden ser agarrados por los workers a medida que estos estén disponibles. Los bloques descargados son agregados al vector de bloques, que lo tiene el nodo (todas las estructuras compartidas están detrás de un Arc<Mutex<T>>, por lo que también se usan de forma segura). Cómo obtener los bloques puede fallar, ya sea porque recibimos un mensaje que no se encontró el bloque o demas, los bundles que no se pudieron descargar son enviados de vuelta al BlockDownloader para que este se los vuelva a mandar a los workers, y el Worker que se encargó de procesar

el bloque que falló, no se encarga de pedir más bloques, porque puede existir el caso que siempre se le pida el bloque que no tiene.
Para terminar la ejecución de los workers, se les manda por el channel un vector vacío representando un End of Channel, los workers interpretan esto y cesan su ejecución, esto se hace para todos los workers que quedaron vivos al finalizar la descarga, de forma que todos tengan un graceful finish.
También puede existir el caso de que no hayan más workers porque se perdieron todos queriendo descargar un bloque que falló, en ese caso el BlockDownloader lee el canal de Missed Bundles en caso que haya quedado algo para descargar, si es asi el mismo BlockDownloader se comunica con un peer para descargar lo que falte, si no hay mas bloques pendientes para descargar se termina la ejecución del BlockDownloader

Comunicación de Nodo con sus peers

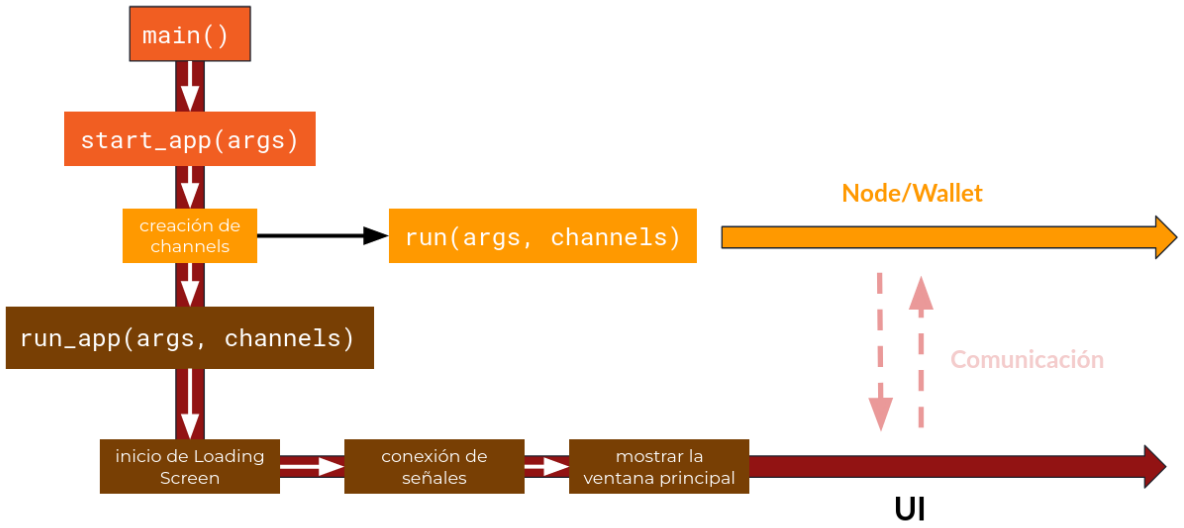
La comunicación se hace también de forma paralela de una forma muy similar a como está construido el BlockDownloader, cada comunicación con otro nodo de la red se realiza en un Thread aparte con la ayuda de una estructura llamada **MessageReceiver**: La diferencia principal con el BlockDownloader es que la comunicación no se hace vía channels, sino que cada thread por separado tiene acceso a los datos compartidos (como el vector de headers o la Blockchain), y estas se encuentran protegidas para que no hayan race conditions a la hora de querer modificar los datos (Arc<Mutex<T>>). El tiempo de vida de cada conexión está dada por un booleano que tiene el nombre de **FinishedIndicator**, que indica cuando los threads tienen que terminar de recibir mensajes, esto es necesario para realizar el Graceful Finish, que una vez haya terminado el thread se hará un join desde el thread principal.

En el Thread principal del Nodo, luego de realizar todas las acciones de inicialización de este, se instancia una Wallet, y luego se queda esperando a recibir solicitudes del usuario (este interactúa con la interfaz gráfica, y esta le manda mensajes a la Wallet según las interacciones del usuario) y se encarga de responderlas adecuadamente. Como la respuesta a los otros nodos se realiza en hilos separados, el hilo principal del programa queda destinado exclusivamente a

esperar mensajes de la Interfaz Gráfica, entonces no afecta al funcionamiento del programa este tiempo de espera para recibir el mensaje.

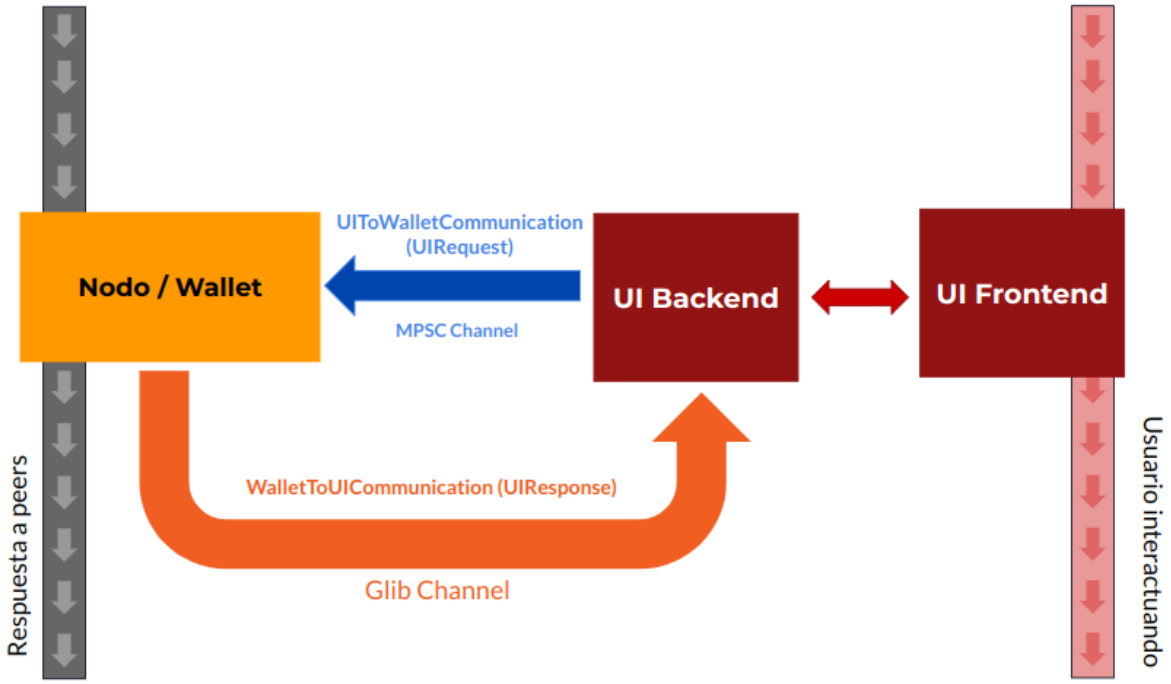
Interfaz Gráfica

La primera implementación del nodo (cuando no había interfaz gráfica) tenía al Nodo corriendo en el thread principal, para esta versión final lo que hicimos fue tener a la UI en el thread principal que se ejecuta cuando hacemos cargo run, y que desde ahí se desprende un thread en donde se ejecute el nodo. El flujo de iniciación de todo el programa es el mostrado a continuación:



main() obtiene los argumentos del programa (que es la ruta al archivo de configuración) y llama a *start_app(args)* que se encarga de inicializar los channels para la comunicación entre el Nodo y la UI, despues de que se origine el thread para el Nodo, empieza la verdadera ejecución de la interfaz gráfica, que en principio tiene una Pantalla de carga que muestra el avance de la inicialización del nodo mostrando las últimas líneas del archivo de log, y cuando se termina de inicializar el nodo se le abre al usuario la ventana principal para poder empezar a interactuar con la billetera. A partir de acá, la UI manda mensajes de actualización cada cierto periodo de tiempo (nosotros decidimos que sea cada 5 segundos) para ver correctamente la información de la billetera en tiempo real.

Ya con la billetera andando, la comunicación entre el Nodo y la UI se muestra a continuación:



La comunicación con el Nodo/Wallet y UI se hace vía el mandado de mensajes siguiendo el protocolo de **ui_communication_protocol.** Usamos dos channels:

- Un **Glib Channel** que comunica al nodo (*sender*) con la UI (*receiver*), el receptor utiliza el método *attach* que solo activa el manejo del mensaje recibido cuando este es mandado por el sender. El backend de la UI se encarga de modificar las cosas para que en el frontend se vea reflejada la respuesta del nodo. Esto se puede ver, por ejemplo, cuando el Nodo responde a cuando queremos cambiar la vista de un bloque desde la pestaña de Transacciones
- Un **MPSC Channel** que comunica a la UI (*sender*) con el Nodo (*receiver*), el nodo en su thread principal queda interactuando con la Wallet y la Wallet recibe los mensajes de la UI, como por ejemplo cambiar de billetera, obtener una Merkle Proof of Inclusion, entre otras.

El Backend de la UI se encarga de asignar las acciones correspondientes a los botones y de comunicarse con el Nodo/Wallet, así como también cargar las

wallets que están guardadas en memoria, mantener actualizada constantemente la wallet, entre otras cosas.

El Frontend de la UI contiene todo lo que el usuario ve y todos los elementos con los que puede interactuar, y desde el Backend nosotros trabajamos con el **Builder** que nos proporciona GTK y de esta forma podemos obtener los objetos de la interfaz desde el backend con los que interactúa el usuario y hacer las acciones necesarias

La lista de respuestas a las solicitudes de la UI son:

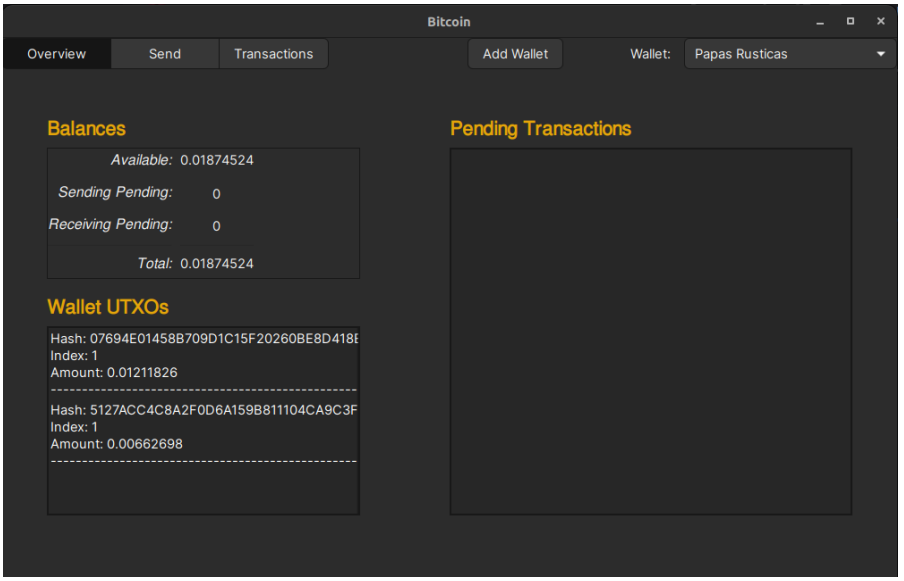
UI Requests To Wallet	Wallet Answers to UI
`ChangeWallet`	`WalletInfo`
`CreateTx`	`TxSent`
`ObtainTxProof`	`ResultOfTxProof`
`LastBlockInfo`	`BlockInfo`
`NextBlockInfo`	`BlockInfo`
`PreviousBlockInfo`	`BlockInfo`
`EndOfProgram`	`WalletFinished`
`UpdateWallet`	`WalletInfo`

Cabe destacar que todos los mensajes pueden devolver un **WalletError** que es manejado de forma adecuada por la UI.

Ventana Principal

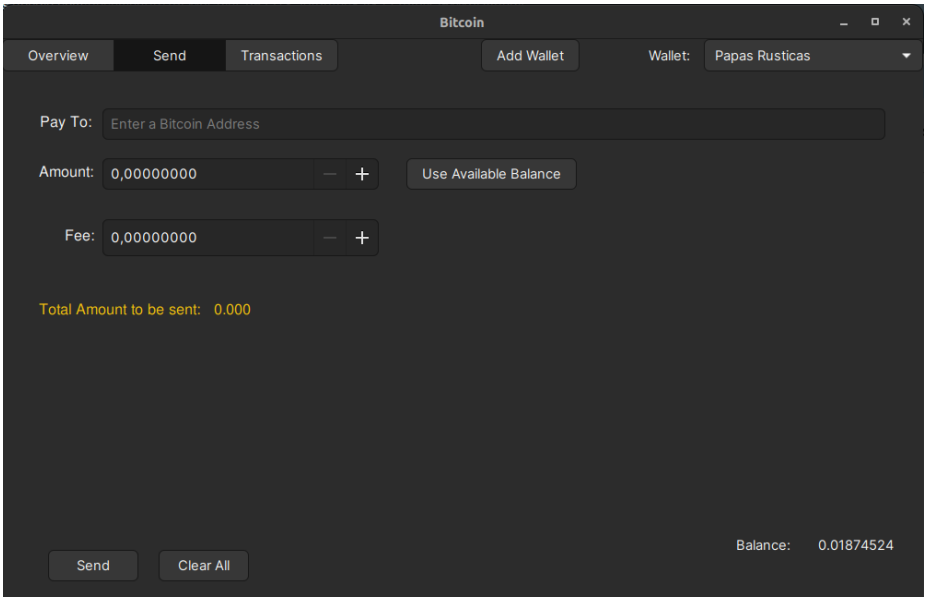
La ventana principal de la aplicación se divide en 3 partes:

Overview



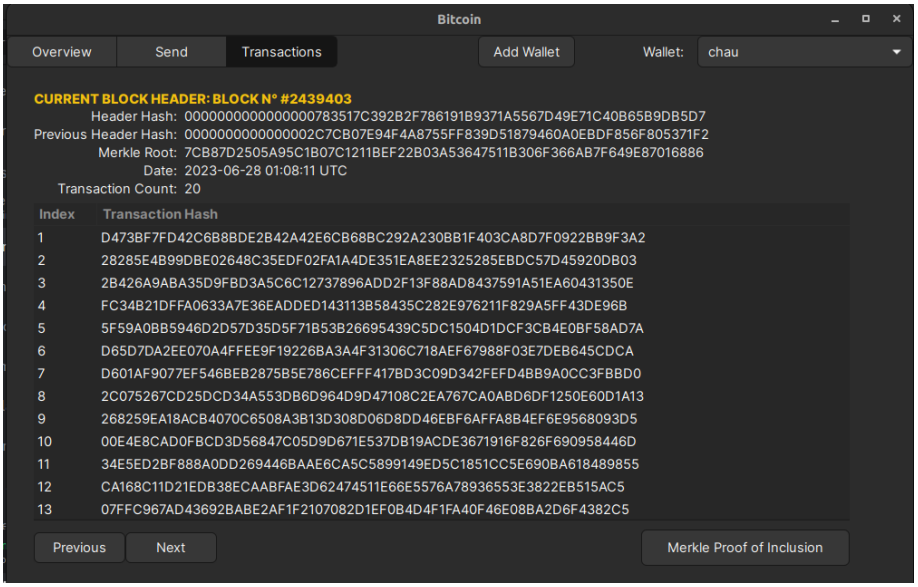
La pestaña de Overview contiene la información de los balances de la billetera, las UTxO para la wallet en cuestión, y las transacciones pendientes de esa cuenta (las que faltan minar). Para todas las pestañas se encuentra el botón para añadir una Wallet y para cambiar de Wallet. Las secciones dentro de la pestaña de overview se actualizan en el momento que la Wallet le manda el mensaje Wallet Info a la UI

Send



La pestaña de Send contiene 2 sliders que permiten poner las cantidad a transferir y la cantidad de fee que el usuario está dispuesto a pagar, todas estas cantidades se muestran en BTC y son manejadas desde el backend en SATOSHIS. Abajo a la derecha hay un indicador del balance, y del otro lado dos botones que se usan para mandar la transacción y para borrar todos los campos.

Transacciones



Esta parte es la que más difiere con respecto a la interfaz de Bitcoin Core, ya que no muestra las transacciones de la propia wallet, su tipo o su fecha, si no que muestra las transacciones que forman parte de los bloques descargados por nuestro nodo. Lo que sí mostramos, es su número y su header, con varios de sus datos. También decidimos añadir un botón que te deja realizar la proof of inclusion en las transacciones que se muestran.