75.42 / 95.08

Taller de programación I

Entrega final

CargoSOS

Padrón:
107677
106005
108225

Autor:
Sofia Javes
Juan Ignacio Biancuzzo
Rafael Francisco Berenguel Ibarra

Fecha:
26/06/23

Sofia Javes
Juan Ignacio Biancuzzo
Rafael Francisco Berenguel Ibarra

Índice

Introducción

En el presente trabajo se pidió el desarrollo de implementación de un Nodo Bitcoin con funcionalidades acotadas que iremos detallando a lo largo del informe.

El trabajo cuenta tanto con los requerimientos funcionales como los no funcionales. Además del presente informe se desarrolló la documentación específica de las funciones, implementaciones especificas y comentarios adicionales sobre el código para un mayor entendimiento de este, el cual se puede obtener con el comando de **cargo doc**.

Se desarrollaron también tests, tanto unitarios como de integración de las funcionalidades más importantes y grandes del trabajo los cuales fueron muy útiles para poder alcanzar el correcto funcionamiento del trabajo.

Se implementó un archivo de configuración, el cual fue de gran ayuda para mantener el código ordenado y evitar los datos hardcodeados. En este se especifican los paths a los archivos de lectura y escritura de los bloques y headers que obtenemos en la descarga. La dirección del DNS, el archivo de escritura de los logs entre otros datos. Junto con esto, se realizó un sistema de loggeo con el fin de que nuestro programa nos comunicara cada acción y respuesta que sucedía al momento de que este estuviera corriendo. El sistema de loggeo fue de gran importancia durante todo el desarrollo del trabajo, y aún más al comienzo, ya que al no tener una interfaz por donde nuestro programa nos comunicara los resultados, lo haciamos a través de los logs.

Asimismo, para un trabajo más ordenado se tomó la decisión de separar y organizar los archivos en diferentes secciones, en otras palabras, carpetas: estructura de nodo, donde se pueden encontrar los archivos en donde se implementan el broadcasting, la descarga de nodos y más. Una carpeta de conexión la cual contiene la conexión a los peers, serialización, la cual se decidió hacer una carpeta de esta característica ya que está presente en muchas otras partes por lo que luego de un refactor se decidió en crear una carpeta propia centrada en la serialización. De esta manera también hay secciones especiales para el sistema de loggeo, para los mensajes y más.

Aunque hablaremos más en detalle de las distintas interfaces desarrolladas, nos parece de gran importancia aclarar la forma en la que se compila el trabajo:

cargo run –bin bitcoin src/bin/bitcoin/nodo_test.conf

Para especificaciones del programa se recomienda la lectura del archivo de configuración donde como mencionamos anteriormente, se puede setear el tipo de interfaz (gráfica o por terminal), si se hará una descarga y lectura a partir de un archivo con información, la cantidad de conexiones que se desean y más.

Arquitectura y diseño

Peer discovery y Handshake

Al empezar el proceso de peer discovery utilizamos una estructura llamada **dns seeder** que se conecta al servidor DNS con una dirección y un número de puerto que se proveen en el archivo de configuración para obtener los Socket Addresses de los nodos que van a ser nuestros posibles peers.

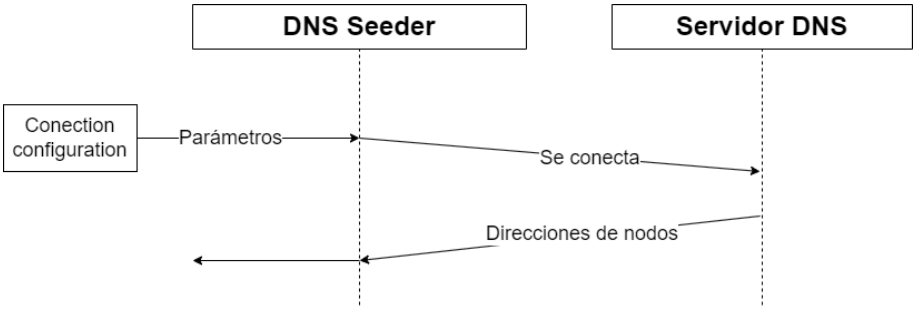

Figura 1: Diagrama del funcionamiento del peer discovery

Junto con el descubrimiento y guardado de los posibles peers, se desarrollaron los primeros mensajes los cuales eran necesarios para el cumplimiento del handshake. Para esto, necesitamos no solo discernir los tipos de mensajes sino enviarlos y poder recibirlos de tal manera que pudieramos entender lo que nos comunicaban y también que los demás lo hicieran. Con esto surgió la necesidad de la serialización y deserealización de los mensajes, la importancia en el orden de cada componente, los tamaños y valores específicos que requería cada mensaje. Para el desarrollo del handshake comenzamos claramente por la implementación de version message y del verack message.

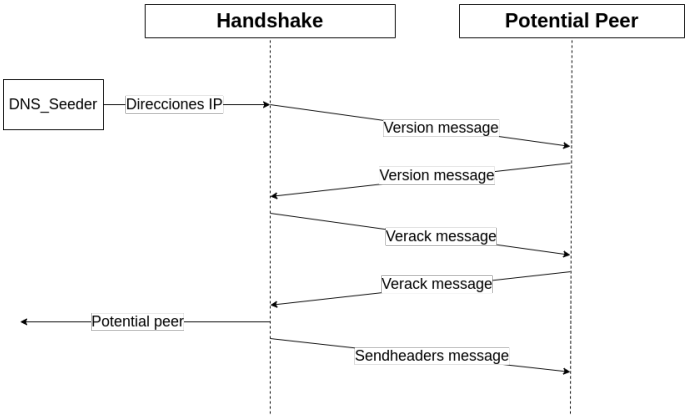

Figura 2: Diagrama del protocolo del handshake

Luego de un intercambio exitoso de version-version para luego enviar y recibir un verack-verack, se podría decir que la conexión con el peer fue exitosa.

Estructura de los mensajes

Al querer comunicarnos con los diferentes nodos tenemos que mandar mensajes que al serializarse tuvieran el formato establecido por el protocolo por lo que decidimos crear una estructura para el header del mensaje como también para su payload.

Para hacer que nuestro código fuera eficiente decidimos que las estructuras que representan los mensajes tuvieran un trait message que obliga a aquellos que la implementan a tener la capacidad de serializarse y deserializarse, e implementa funciones que se repiten en todos los mensajes.

De esa forma tenemos un mensaje que por un lado tiene el header y por otro la estructura que implementa este trait.

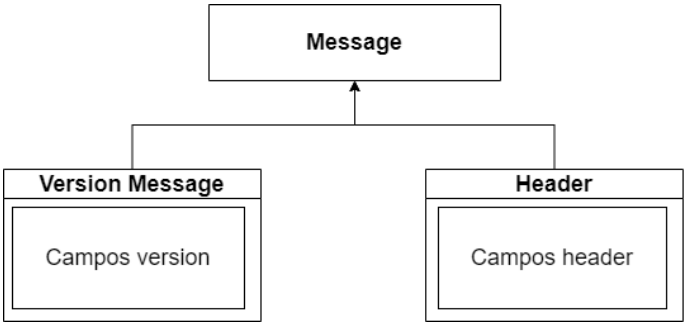

Figura 3: Diagrama de la estructura de mensajes

Initial Block Download

Una vez finalizado el handshake y teniendo peers que nos reconocen, damos inicio al proceso de initial block download.

Nuestro programa, por recomendación de la cátedra, implementa el protocolo headers first para hacer la descarga inicial de bloques. Esta se puede dividir en dos partes, el initial headers download y el block download en sí.

Tenemos una estructura llamada **InitialHeaderDownload**, la cual se encarga de crear y mandar un mensaje del tipo **get headers**, el cual debe contener un hash del header del último bloque que tenemos en nuestra blockchain. Serializamos este mensaje y elegimos uno de nuestros peers para mandarle el mensaje.

Este peer nos va a responder con un mensaje del tipo **headers**, el cual contiene serializados un máximo de 2000 headers.

Nosotros deserializamos ese mensaje, guardando todos los headers en nuestra blockchain.

Cuando recibimos menos de 2000 headers entonces sabemos que tenemos que parar y empezamos a pedir los bloques que le corresponden a los headers.

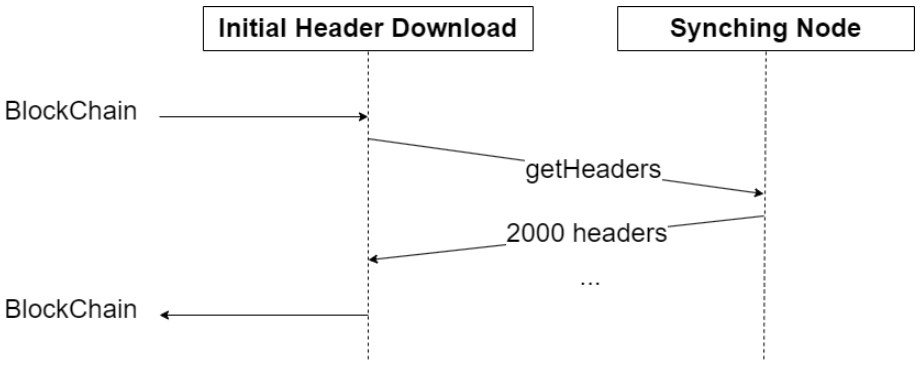

Figura 4: Diagrama del protocolo del Initial Block Headers

La estructura encargada de pedir los bloques se llama **BlockDownload**. Esta estructura se encarga de crear y mandar a un peer un mensaje del tipo **GetData** el cual pide todos los bloques con los headers que conseguimos para luego actualizar la blockchain con ellos, hasta haber conseguido todos los bloques que necesitaba.

Una vez que hemos descargado todos los headers y sus respectivos bloques, consideramos que ha finalizado el initial block download.

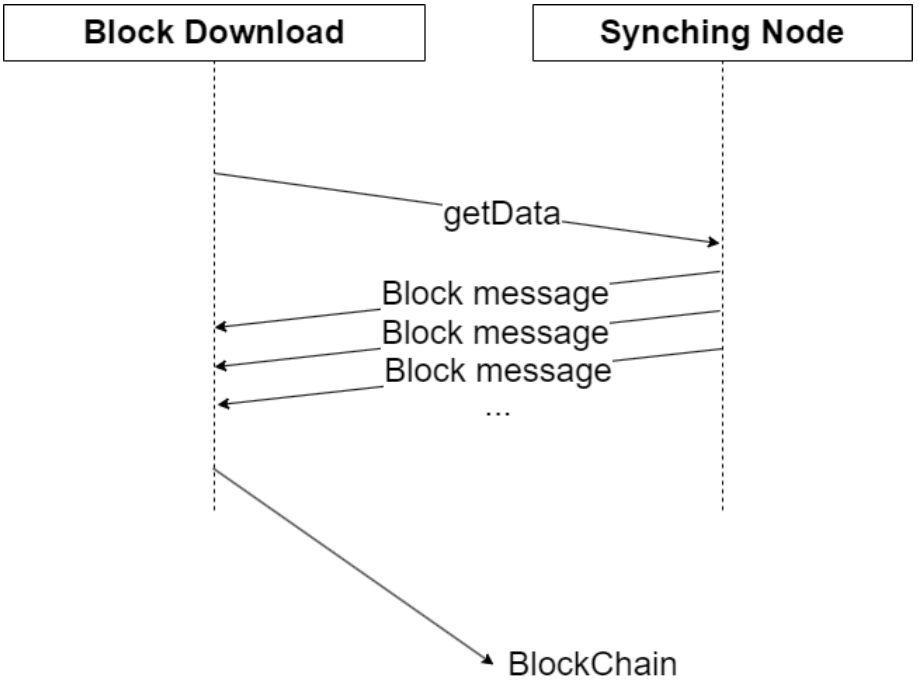

Figura 5: Diagrama del protocolo del Block Download

BlockChain

Para la descarga se realizó una propia implementación de la blockchain para poder guardar la información de los nodos.

En este caso la estructura de la blockchain contiene un vector de nodos de la block chain llamados nodechain, y también tendrá una referencia a los últimos bloques donde se hizo un fork. Cada **NodeChain** contendrá efectivamente información de un bloque, el hash del header y el índice del nodo previo a este.

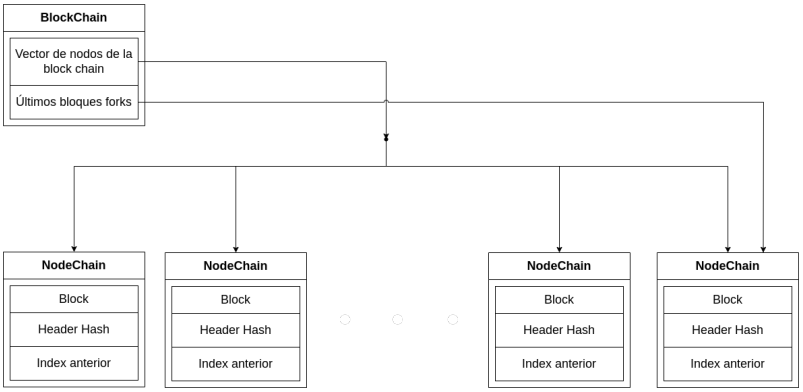

Figura 6: Estructura definitiva de la blockchain

Manejo de mensajes

Para la descarga de headers y bloques tuvimos que implementar más mensajes tal como el **getHeaders** el cual se enviaba para la obtención de headers a los peers, pero como respuesta a este se implementó el headers message, el cual recibimos como máximo 2000 headers para descargar.

Asimismo también para la descarga de bloques se implementaron los mensajes de **getData** y mensajes de bloque para poder recibirlos y descargarlos los cuales una vez hecho esto, creamos nuestra blockchain.

Para este entonces al tener que serializar y deserealizar cada uno de los mensajes se decidió hacer un trait para que los mensajes y cada tipo de datos que estos contuvieran fueran tanto serializables como deserealizables.

Además de que sabríamos que luego tendríamos que implementar más mensajes para el broadcast, la recepción y el envío de transacciones y más.

Comportamiento del nodo

A continuación se detallará el comportamiento reducido que implementamos para este trabajo.

Wallet

La wallet es la estructura en la que decidimos guardar nuestras diferentes cuentas de bitcoin.

Wallet		
Account 1	**Account 2**	**Account 3**
account_name: Juan	account_name: Sofi	account_name: Rafa
private_key	private_key	private_key
public_key	public_key	public_key
address	address	address

Figura 7: Estructura de la wallet

Es una estructura que almacena muchas cuentas y se encarga de administrarlas, ya sea agregando nuevas cuentas o borrándolas, así como también es la encargada de proveer datos generales como la cantidad de cuentas o si tenemos alguna en específico disponible. También tiene siempre una cuenta seleccionada, lo cual es útil para utilizar en conjunto con las interfaces.

Lo que almacena la wallet son las **accounts**, que son las estructuras que representan cuentas de bitcoin. Estas tienen a su disposición la información de la cuenta, como lo es un nombre que nosotros le demos para poder identificarla, la llave privada de la cuenta, que es la llave que se utiliza para poder firmar información; la llave pública de la cuenta, que utilizamos para crear secuencias de bytes que constaten que somos los que firmaron algo con nuestra llave privada; y también el address de la cuenta, que es el principal medio por el cual se identifica a una cuenta y que se obtiene hasheando la llave pública. Para la estructura de la llave privada hicimos uso de la crate de rust **secp256k1**, pues esta tenía la funcionalidad para las firmas de curva elíptica.

A través de estas diferentes estructuras de datos, la cuenta tiene diferentes funcionalidades, que permiten verificar si esta es la dueña de una transacción, o una de las transacciones de un bloque de transacciones y poder crear y firmar transacciones con los fondos que tiene disponibles.

También cabe destacar que la wallet, que almacena todas estas cuentas, es capaz de serializarse en un archivo para poder ser cargada al correr el programa en instancias posteriores, lo cual permite recuperar la información de nuestras cuentas sin la necesidad de ingresar nuestras llaves públicas y privadas nuevamente.

Utxo set

El utxo set es la estructura encargada de mantener un registro de los transaction outputs que todavía no han sido gastados.

Internamente tiene un diccionario que tiene como llave el Outpoint que tiene la información de la posición del output disponible y como valor el output en sí.

Tiene funcionalidades, como las de crearse a partir de una blockchain obteniendo los outputs que no han sido gastados, actualizarse con nuevos bloques que vamos obteniendo y poder calcular el balance de un cuenta que le especifiquemos.

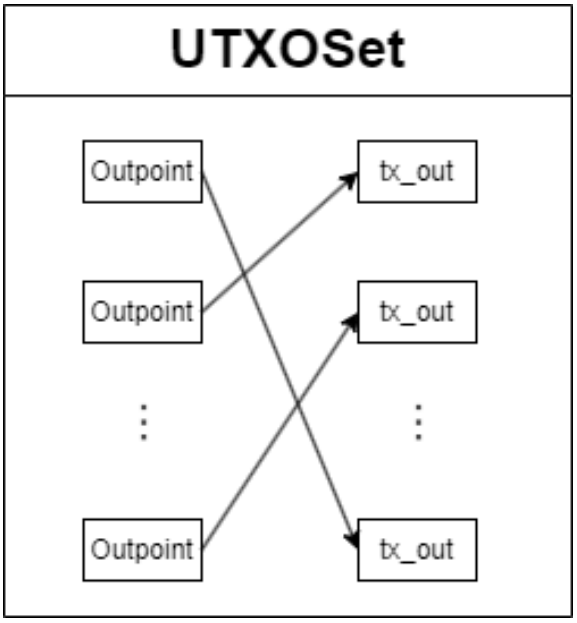

Figura 8: Estructura deL UTXOSet

Broadcasting

Hacer referencia al broadcasting es hacer referencia a la comunicación más en detalle con los peers. En este caso no es de una forma estrcturada tal como lo fue el handshake o la descarga inicial sino la comunicación constante con cada uno de los peers con los que logramos establecer una conexión.

En esta oportunidad optamos por la solución más simple que pudimos encontrar la que fue delegar la responsabilidad de la estructura del broadcasting de enviar transacciones a los peers y recibir los mensajes que estos nos enviaran ya fuera como respuesta a estos mensajes que enviamos como también las notificaciones de nuevos bloques y transacciones.

Creación de transacciones

La manera en la que implementamos la creación de transacciones consiste en primer lugar, a través de nuestro utxo set, obtener los transaction outputs a nombre de la cuenta que queremos que mande la transacción.

Si tenemos suficientes fondos, entonces creamos los transaction input que gastan nuestros outputs y también creamos los transaction outputs, uno para pagar a la cuenta a la que le queremos transferir bitcoin, y la otra para darnos el 'cambio'. La diferencia entre la suma de los inputs y la suma de los outputs es lo que se llevará como comisión el minero que ponga esta transaction en el bloque

que está intentando minar. Una vez que hemos creado nuestra transacción hacemos uso de la llave privada para firmarla creando los signature scripts.

Envío de transacciones

Una vez que tenemos nuestra transacción firmada, podemos enviarla a todos los peers con lo que mantenemos una conexión en forma de un mensaje **tx**. Estos al constatar que nuestra transacción es válida la enviarán a aquellos (otros) peers con los que mantengan conexiones haciendo que la transacción efectivamente sea broadcasteada.

Recepción de transacciones

Los peers a los que estemos conectados nos pueden mandar transacciones de dos manera distintas. La primera es a través de un mensaje **tx** suelto que contenga una transacción serializada como payload. La otra manera (y la más común) es que nos manden un mensaje del tipo **inv** en donde nos den el hash de una nueva transacción para que respondamos con un mensaje **get data** pidiendo esa transacción, la cual es finalmente mandada a través de un mensaje tx.

Interfaz

Se implementó una interfaz para que el usuario pueda interactuar en cercanía con el programa. Se desarrolló una interfaz tanto por terminal como gráfica. Ambas tienen las mismas funcionalidades, carga de cuentas (ver especificaciones tal como balance y transacciones recientes), envío y recepción de transacciones.

Interfaz por Terminal (TUI)

La interfaz por terminal se intenta que sea de los más amigable al usuario, siempre comunicando cada mensaje y requerimiento para cada opción que se quiera seleccionar, ya que hay un menu display con diferentes opciones. En este caso para ingresar datos el usuario lo hace a través de la terminal.

La interfaz por terminal tiene la posibilidad de guardar las cuentas que el usuario ingresa, se pueden ver las especificaciones tal como el balance, se pueden enviar transacciones y recibirlas. Al tener la habilidad de guardar diferentes cuentas el usuario puede seleccionar alguna de estas. El diseño se intenta que sea de lo más simple posible para un claro entendimiento desde el lado del usuario.

Interfaz Grafica (GUI)

Para la implementación de la interfaz gráfica se utilizó **gtk3** para poder usar en conjunto el diseñador de interfaces **Glade** que fue de gran ayuda para la generación del código XML para darle el diseño gráfico que queríamos obtener. Nos basamos en las sugerencias de la cátedra con las capturas de ejemplo que se mostraron como también de la aplicación de Bitcoin Core.

Mediante la aplicación gráfica el usuario puede: crear una nueva cuenta, guardas varias cuentas, obtener datos de esta, tal como el cálculo del balance como un display de las transacciones que involucran a la cuenta actualmente seleccionada. Se puede enviar transacciones con un monto a elección del usuario teniendo en cuenta a libre elección también el fee con el cual se hará la transacción. Se intentó en todo momento que el manejo de errores fuera lo más claro posible ya que al tener uno saltan ventanas de notificación que especifican cual fue el error. Un ejemplo de esto serìa intentar enviar una transacción sin ingresar el address.

En este caso para lograr un correcto funcionamiento entre el front y el back se decidió la implementación de 2 canales, uno para cada parte. Cada canal contando con un sender y un receiver. Además como se necesita que el front se actualice al momento de algún cambio por parte del back, se utilizó un canal especial de MainContext el cual permite recibir las señales siempre que sea necesario evitando así el 'desfasaje' entre front y back y que el usuario esté al tanto de cada cambio en el menor tiempo posible en el que suceda.

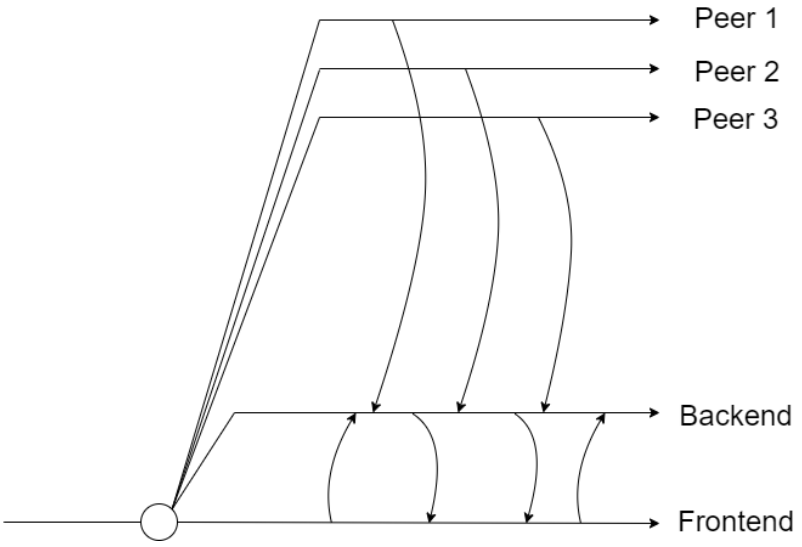

Figura 9: Diagrama de los threads cuando usamos la gui

Concurrencia

A lo largo del trabajo se hizo uso de la concurrencia. Nos vimos obligados a implementar diferentes hilos aparte del principal para evitar el funcionamiento secuencial de las tareas. Además de que en muchos casos esto fue necesario ya que necesitabamos un constante funcionamiento de algunas partes mientras se realizan otras. Tal como en el broadcasting, para poder asegurarnos que una transacción fue enviada y agregada a un bloque deberíamos estar recibiendo y manteniendo conexiones con los peers constantemente.

En el caso del cierre del programa se utiliza la función implementada **destroy** la cual destruye el programa. No sin antes manejar todos los recursos en uso y asegurandose de cerrarlos. En este caso se realiza un **join** por cada peer con el que mantenemos una conexión. Una vez hecho esto con todos los peers, se cierra el programa definitivamente.

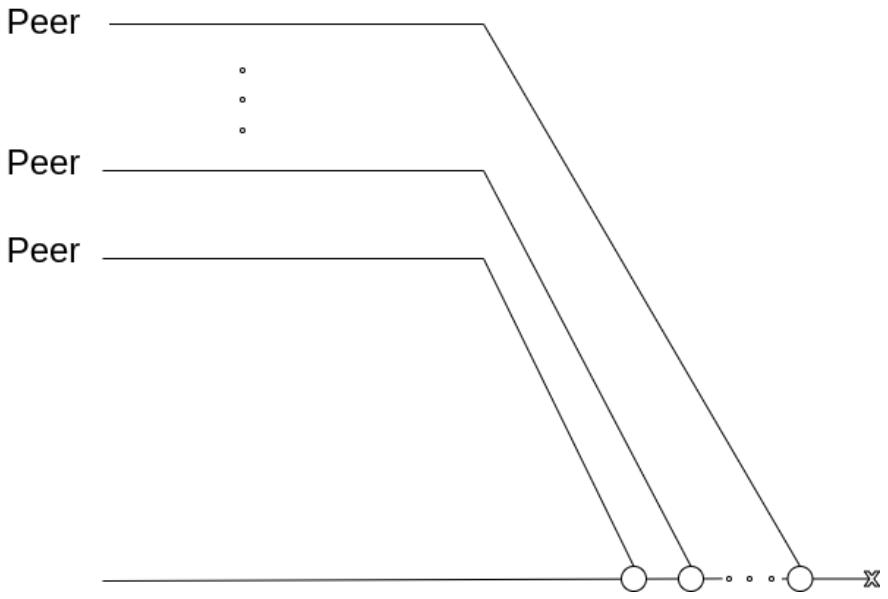

Figura 10: Finalización y cierre del programa

Conclusión

Luego del extenso trabajo que se desarrolló a lo largo de la cursada podemos decir que los conocimientos aprendidos fueron de gran importancia. Recordando en un primer momento al comienzo de la cursada donde no teníamos conocimiento alguno del lenguaje de Rust, ahora en la actualidad podemos decir que logramos desarrollar un programa de gran envergadura y conseguimos los objetivos principales del trabajo utilizando las especificaciones requeridas tanto del lenguaje como de la materia.

Se priorizó siempre la organización y la buena modularización para que el código fuera de fácil entendimiento a quien tuviera que leerlo., siendo de gran ayuda también la documentación. Al igual que esto, se dio mucha importancia a la organización trabajo en equipo, lo que fue muy útil ya que las oportunidades en las que el programa tuvo conflictos fueron casi nulas permitiendo así un trabajo fluido y ameno para cada uno de nosotros. Todas las etapas e iteraciones por las que se pasó con el trabajo se pueden ver documentadas en el repositorio del trabajo.