Design and implementation of a Bitcoin Simulator

Víctor Mora Afonso
Consultora: Cristina Pérez Solà
Master Interuniversitario de Seguridad de las Tecnologías de la Información y las Comunicaciones
Universitat Rovira i Virgili - Universitat Oberta de Catalunya – Universitat Autónoma de Barcelona
Trabajo de Fin de Master

Santa Cruz de Tenerife, 13 de Junio de 2014

Agradecimientos

Si ya es difícil sacar un proyecto adelante, todavía lo es más compaginando los estudios con el trabajo y con mi rama emprendedora. Por eso, en primer lugar me gustaría agradecer a amigos, familiares y compañeros que han aportado su granito de arena con palabras de motivación o simplemente una sonrisa a cambio de charlas sobre Bitcoin o seguridad informática.

En segundo lugar me gustaría agradecer a Cristina, que a pesar de la distancia ha estado bien cerca, motivándome con cada idea o sugerencia que aportaba y con cada paso que iba dando.

Gracias.

Resumen

Bitcoin es una de las primeras implementaciones exitosas de una cripto-moneda. A diferencia de las monedas convencionales, Bitcoin carece de una autoridad centralizada encargada de la emisión de monedas y del control, sino que su funcionamiento se basa en una red peer-to-peer. Aunque es difícil dar números exactos del uso del sistema, se estima que en 2011 había 60,000 usuarios de Bitcoin y que el total de bitcoins en la red es equivalente a un total de 110 millones de dólares.

En este trabajo se presenta un simulador de Bitcoin, es decir, un software capaz de reproducir el comportamiento del sistema Bitcoin y que permite estudiar cómo diferentes perfiles de usuario afectan al rendimiento del sistema o si es posible ejecutar ciertos ataques teóricos a la red en un escenario real. Se ha desarrollado un simulador capaz de reproducir el comportamiento de la red Bitcoin durante largos periodos de tiempo en instantes, tomando cuenta de todas las interacciones llevadas a cabo como intercambio de mensajes o generación de bloques. El simulador cuenta con una interfaz web desde la que se pueden configurar y lanzar las simulaciones así como realizar un análisis posterior de los resultados a partir de una serie de gráficas y visualizaciones de todo lo acontecido.

Palabras clave

Bitcoin, Criptomoneda, Simulador de red, Criptografía.

Abstract

Bitcoin is a digital currency based on the use of strong-cryptography. It is one of the first implementations of a crypto-currency. Unlike most conventional currencies, Bitcoin has no central issuing authority and uses peer-to-peer technology to operate. Although it is difficult to provide exact numbers of the system, estimations suggested that there were about 60,000 users in the network in 2011 and that the total amount of bitcoins is equal to over 110 million USD. In this work it is presented a Bitcoin simulator, that is, a piece of software able to reproduce the Bitcoin system behavior which allows to study how different user behaviors affect the global system performance or if it is feasible to perform certain theoretical attacks on the network. The developed simulator is able to reproduce the network activity during long periods of time in a few seconds, taking into account any message exchange or block generations. The simulations can be configured and launched from a user friendly web interface which also allows users to analyze results providing different charts and data visualizations.

Keywords

Bitcoin, Crypto-currency, Network simulator, Cryptography.

Índice general

Índice de figuras

Índice de tablas

1 Introducción

1.1 Motivación

Desde su aparición y en especial en el último año Bitcoin ha pasado de ser una cripto-moneda más a pasar a estar en boca de todos. Su uso ha ido creciendo de forma exponencial hasta lo que es hoy en día y en apenas cuatro años de vida en los que el valor de la primera transacción realizada para comprar dos pizzas por 10,000 bitcoins (aproximadamente 25$) a fecha de hoy sería equivalente a 5,930,000$. Parece que Bitcoin está aquí para quedarse.

Desde su llegada, se ha ido creando a su alrededor un ecosistema de tiendas, mercados, casinos o casas de cambio que solo trabajan con Bitcoin. Existen hasta cajeros automáticos de Bitcoin. Cada vez son más los comercios tradicionales que aceptan esta moneda como pago por no sólo servicios sino también productos físicos. [1]

Son las propias características de Bitcoin las que han posibilitado su crecimiento tan rápido a su vez que han generado una gran controversia. Toda transacción en Bitcoin es anónima y no existe una autoridad central encargada de la generación de monedas o del control de las transacciones. Esto ha permitido que Bitcoin sea perfecto para el desarrollo de actividades ilegales sin que se pueda ser detectado. Un ejemplo de esto es Silk Road, recientemente cerrado por el FBI, que llegó a convertirse en el mayor portal de tráfico de narcóticos de todo internet.

El tráfico libre que permite Bitcoin ha hecho que su uso se haya disparado en países con grandes restricciones y controles de capital y sobre las divisas como China, cuyo banco central recientemente ha publicado diversos documentos con intención de restringir el uso de la moneda en el país. La controversia no termina aquí, sino que también numerosos operadores de Bitcoin han sido víctimas de potentes ataques distribuidos de denegación de servicio. [2]

Sin duda alguna, una de las características que más llama la atención del funcionamiento de Bitcoin es su protocolo, que está basado en una red distribuida. La principal diferencia con otras cripto-monedas predecesoras de Bitcoin es que todas estas requerían de una autoridad central encargada del control de transacciones y de preservar el anonimato de los usuarios. Bitcoin no requiere de ningún tipo de autoridad, sino que la propia red se gobierna “sola” siguiendo unas reglas y se garantizará su buen funcionamiento mientras la mayoría siga esas reglas. Aunque la clave de su éxito gira en torno al proceso de minado y las recompensas que reciben los usuarios que colaboran con la red Bitcoin, que atrajo a muchos usuarios que comenzaron a utilizarla convirtiéndola en lo que es hoy en día.

El protocolo Bitcoin hace uso de diferentes primitivas criptográficas como funciones hash, firmas electrónicas o curvas elípticas que lo hacen bastante atractivo y elegante desde el punto de vista de la criptografía y la seguridad informática.

1.2 Objetivos

Para el desarrollo de este trabajo se marcó como objetivo principal diseñar un simulador de Bitcoin capaz de emular los elementos más importantes del protocolo Bitcoin como es el proceso de minado, es decir, la creación y adición de bloques a la cadena principal teniendo en cuenta los aspectos y características de una red de datos real en la que intervienen factores como la latencia o el ancho de banda. Para ello era necesario estudiar en profundidad el funcionamiento del protocolo Bitcoin y de su cliente, ya que el software desarrollado intentaría emular el comportamiento de estos componentes haciendo necesario tener un control y conocimiento de todas las funcionalidades y pormenores para poder dar lugar un producto más fiel a la realidad. A su vez, el objetivo propio de implementar un simulador es

el de observar el comportamiento de sus componentes ante ciertas teorías que sería muy complicado comprobar en la realidad, por lo que se hacía necesario también antes de comenzar el diseño del simulador el estudio de algunos ataques que se hayan propuesto contra la red para valorar su eficacia.

El conjunto de objetivos concretos de este trabajo se pueden desglosar en los siguientes:

- Estudio en profundidad del funcionamiento del protocolo Bitcoin y del cliente Bitcoin para poder reflejar sus funciones y actividades en el simulador.
- Estudio y comprensión de diferentes ataques que se pueden llevar a cabo contra la red, el protocolo o el propio cliente Bitcoin para una vez analizados poder ejecutarlos dentro del entorno controlado del simulador.
- Análisis y diseño de la estructura y arquitectura del simulador. Conceptualización, análisis contextual de tareas, objetos, plataforma.
- Codificación y desarrollo del simulador.
- Diseño y desarrollo de un módulo de visualización de los datos generados por el simulador para un posterior análisis.
- Realización de diferentes pruebas del simulador con algunos de los ataques y comportamientos estudiados.

1.3 Fases del desarrollo

El desarrollo de este trabajo se ha dividido en tres fases principales y bien diferenciadas.

En la primera fase el trabajo estuvo enfocado a la investigación, primero sobre la arquitectura y usos de Bitcoin, y luego sobre ataques y contramedidas, para poder tener los suficientes conocimientos que permitirán diseñar un simulador acorde a las características del protocolo.

Una vez concluida la fase de investigación comenzó la fase de desarrollo del simulador. Para ello hubo que diseñar la arquitectura del mismo y definir todos sus componentes, para una vez se tenía claro lo que había que desarrollar podar dar comienzo a la etapa de codificación. Para ello fue necesario aprender el lenguaje Python, lenguaje en el que está desarrollado el simulador y del que no se tenían conocimientos antes del comienzo de este trabajo.

Una vez se tenía un simulador funcional, podía dar comienzo la fase de desarrollo del módulo de visualización, que fue desarrollado en una aplicación web y que permitió observar diferentes comportamientos de la red del simulador en función de las acciones y características de los nodos.

1.4 Estructura de la memoria

El contenido restante de esta memoria está organizado de la siguiente manera:

- Capítulo 2. Se introduce el protocolo Bitcoin, presentando una definición formal del mismo. Se habla de su historia y evolución desde su nacimiento, y se hace una descripción de los principales casos de uso y ventajas que puede ofrecer a los usuarios.
- Capítulo 3. Incluye un estudio de ataques que se han propuesto o se han llevado a cabo contra el protocolo Bitcoin.
- Capítulo 4. Describe el comienzo de la fase de desarrollo del trabajo incluyendo el diseño, la estructura, arquitectura y características del simulador así como de la interfaz desde la que controlar la simulación y analizar los resultados.
- Capítulo 5. Detalla la implementación del simulador, los componentes utilizados, problemas afrontados y cambios en el diseño inicial del mismo.
- Capítulo 6. Detalla la implementación del servidor y aplicación web para el control y visualización.

- Capítulo 7. Se presenta un análisis de diferentes comportamientos de la red en el simulador utilizando distintas configuraciones de los nodos.
- Capítulo 8. Recoge las conclusiones del trabajo, y posibles mejoras.

2 Preliminares sobre Bitcoin

2.1 Definición

Bitcoin se trata de una moneda digital descentralizada que permite la realización de pagos instantáneos a cualquier persona en cualquier lugar del mundo. Bitcoin utiliza protocolos "peer-to-peer" para poder operar sin la necesidad de una autoridad central. La gestión de las transacciones y la emisión de las monedas es llevada a cabo por la red de Bitcoin de forma colaborativa.

Bitcoin se trata de la primera implementación exitosa de una cripto-moneda distribuida. Teniendo en cuenta que dinero es cualquier objeto o tipo de registro aceptado como pago a cambio de bienes y servicios en un determinado país o contexto socio económico, Bitcoin está diseñado en torno a la idea de usar criptografía para controlar la creación y transferencia del dinero en lugar de confiar en una autoridad central.

El diseño de Bitcoin comenzó en torno al año 2008 cuando una persona o grupo de personas bajo el pseudónimo de Satoshi Nakamoto publicó un artículo describiendo esta cripto-moneda [3]. En el año 2009 se lanzó la primera versión del cliente Bitcoin [4] que inició la red y creó las primeras monedas.

Para poder entender Bitcoin, su funcionamiento y las ventajas que ofrece frente a los sistemas de pago tradicionales hay que definir una serie de conceptos básicos:

- Moneda. Suponiendo que A quiere comprar un objeto que B tiene a la venta. Para llevar a cabo este objetivo, A deberá entregar a B algo de igual valor. La forma más sencilla de hacer esto es utilizando un objeto de cambio que B acepte. Esto es lo que se conoce como moneda. Este sistema hace que el comercio sea más fácil eliminando la necesidad de coincidencia de necesidades como en otros sistemas de comercio como el trueque. Una moneda y su ámbito de aceptación puede ser global, nacional o local.
- Banco. A no necesita entregar la moneda a B en persona, sino que en su lugar puede transferirla a través de un intermediario que se encarga de almacenar y proteger las monedas de A, un banco. A puede retirar del banco tanto dinero como haya depositado. Normalmente un banco, cobrando cierta comisión, se encargará de enviar el dinero a B en su nombre. Para ello se pondrá en contacto con el banco de B y realizará una transacción. Mediante el uso de cajeros automáticos o en una sucursal bancaria, un cliente puede retirar dinero, realizar transacciones o consultar el dinero que tiene en sus cuentas y confían en que la cantidad de dinero que aparece en las pantallas o recibos es la cantidad de dinero que pueden extraer del banco. Están tan seguro de ello que llegan a aceptar estos números de la misma manera que aceptan billetes.

Bitcoin se trata de un sistema de comercio en el que se poseen y transmiten cantidades de bitcoins entre los diferentes usuarios del sistema de forma parecida al proceso descrito arriba pero ofreciendo una serie de ventajas con respecto a la banca online, abaratando los costes por transacción al eliminar cualquier intermediario y autoridad central, y garantizando el anonimato de los usuarios.

Como Bitcoin se trata de una moneda y de un protocolo es necesario establecer una serie de convenciones para evitar confusión. Normalmente se utiliza el término Bitcoin para referirse al protocolo y el término bitcoins (en minúscula) para referirse a una unidad de la moneda Bitcoin. La unidad utilizada para representar los bitcoins es BTC.

A día de hoy hay prácticamente 13 millones de bitcoins en circulación [5] con un precio de mercado de 650 dólares [6], se realizan alrededor de 70.000 transacciones diarias [7], habiéndose realizado un total de 40 millones de transacciones [8].

2.2 Características

Normalmente, un sistema de banca tradicional presenta una serie de desventajas que pueden hacerlos menos deseables:

- Es costoso. La comisión por realizar una transacción puede llegar a costar un porcentaje bastante elevado de la cantidad a transferir. [9]
- Es lento. Una transacción entre dos entidades bancarias diferentes puede tardar varios días en realizarse y confirmarse.
- En la mayoría de los casos se trata de un sistema en el que no se garantiza el anonimato.
- Una cuenta puede ser congelada o incluso todo su saldo puede ser confiscado.
- Algunos sistemas de pago pueden negarse a aceptar pagos de ciertas entidades o desde ciertos países [10].

Bitcoin presenta una serie de características que hace que usuarios como compradores y comerciantes vean en este sistema una serie de ventajas con respecto a los sistemas de banca tradicional:

- Descentralizado. Ninguna autoridad central controla Bitcoin. Las nuevas monedas son creada mediante un algoritmo y son entregadas a aquellos que aportan tiempo de computación para ayudar a mantener las transacciones seguras. Todo el mundo tiene una copia de la base de datos de las transacciones y de todas las reglas y todo el mundo verifica que las transacciones son correctas.
- Anónimo. Toda cuenta de bitcoin es una pareja de clave pública / privada. La clave pública se puede entender como un número de cuenta al que se le puede enviar dinero. No hay manera de mapear estos números de cuenta a identidades, pero sí que es posible trazar las transacciones de una determinada cuenta al tener todo el mundo el histórico de las transacciones. Se puede evitar esta trazabilidad utilizando cuentas diferentes para cada transacción.
- 21 millones de bitcoins. Este es el número máximo de unidades que existirán y está limitado por el diseño de la red, lo que hace que al ser un bien limitado su valor vaya aumentando. Un bitcoin se puede fraccionar hasta 8 posiciones decimales.
- Transacciones irreversibles. Una transacción no se puede cancelar o devolver. La única manera sería que el receptor decidiera voluntariamente devolver el pago mediante una nueva transacción.
- No hay comisiones por transacciones. A diferencia de los pagos con tarjeta de crédito, en los que al comerciante se le cobra una cierta cuota por cada transacción, esto no ocurre en Bitcoin independientemente de la cantidad que se vaya a transferir. Sí es cierto que se pueden incluir de forma voluntaria incentivos para conseguir que una transacción se realice antes.
- Libertad de pagos. Es posible enviar y recibir cualquier cantidad de bitcoins prácticamente de forma instantánea a cualquier lugar del mundo en cualquier momento. No hay fronteras y los usuarios de Bitcoin tienen un control total sobre su dinero.
- Transparente y neutral. Toda la información relativa a Bitcoin está disponible a todo el mundo y cualquiera puede utilizarla para realizar cualquier consulta o verificación. Además, el protocolo Bitcoin utiliza numerosas primitivas criptográficas haciendo que sea seguro y confiable a la vez que transparente y predecible.
- Seguridad y control. Los usuarios de Bitcoin tienen un control total sobre sus transacciones. Es imposible por parte un comerciante forzar un cobro sin el consentimiento de los usuarios.

A su vez, Bitcoin presenta una serie de desventajas propias de su corto tiempo de vida:

- Grado de aceptación. No todo el mundo conoce la existencia de esta moneda. Cada día son más y más los negocios que aceptan bitcoins como moneda de pago, pero aun así esta lista es pequeña [11].
- Inestabilidad. El número total de bitcoins en circulación y el número de negocios utilizándolo es demasiado pequeño con respecto a lo que podría llegar a ser. Es por esto que el valor de mercado de la moneda fluctúa con mucha facilidad [6] haciendo que el precio de un producto o servicio se vea afectado significativamente. Esta volatilidad irá descendiendo a medida que el mercado Bitcoin madure y se asiente.
- Desarrollo en proceso. El cliente Bitcoin todavía se encuentra en fase beta con algunas de sus características incompletas y en desarrollo. Cada día se implementan nuevas herramientas y servicios para hacer el sistema más seguro. Bitcoin acaba de nacer y está en proceso de maduración.

2.3 Funcionamiento

Antes de poder explicar los detalles técnicos del simulador de Bitcoin en los próximos capítulos, es necesario entender su funcionamiento desde el punto de vista de un usuario. Según el ejemplo del banco explicado anteriormente, en Bitcoin se definen dos procesos fundamentales, la creación de monedas y la realización de pagos.

Creación de monedas

En un sistema de comercio, el proceso de creación de monedas debe ser limitado de alguna manera para estas puedan tener un valor. En Bitcoin las nuevas monedas son generadas a través de un proceso denominado “minado” siguiendo una serie de reglas acordadas por la red. Un usuario para minar bitcoins ejecutará un programa que estará continuando buscando la solución a un problema matemático difícil de resolver y cuya dificultad es conocida. Esta dificultad es ajustada por la red periódicamente de manera que el número de soluciones encontradas de manera global sea constante independientemente de la capacidad de cómputo de la red. Cuando se descubre una nueva solución, esta es anunciada a todo el mundo junto con más información empaquetada en lo que se denomina un bloque. En la red se intenta que el número de bloques generados sea de 1 cada 10 minutos. La generación de bloques es un proceso importante dentro de Bitcoin, ya que es el proceso utilizado para la validación de las transacciones.

Cada vez que se genera un nuevo bloque, se genera una cantidad de bitcoins predeterminada con la que el usuario descubridor del bloque puede hacer lo que quiera (normalmente se la adjudicará). Esta recompensa es un incentivo para que los usuarios colaboren con la red aportando su capacidad de cómputo y poder generar nuevos bloques. En la actualidad el número de bitcoins ofrecidos al usuario que obtiene un bloque es de 25 BTC, aunque esta cantidad no es fija, sino que se divida a la mitad cada 210000 bloques, de manera que llegará un momento en que no se generarán nuevas monedas, llegando al total de 21 millones de bitcoins que existirán en total. Puesto que esta recompensa irá disminuyendo a lo largo del tiempo, llegará un momento en que la compensación por el uso de hardware y electricidad de los “mineros” será recogida a partir de las comisiones de las transacciones.

Realización de pagos

Bitcoin utiliza criptografía de clave pública y firmas para garantizar la autenticidad de los usuarios y que un atacante no pueda utilizar el dinero de otros usuarios.

Cada usuario del sistema tendrá una o más “cuentas” asociadas cada una de ellas a una pareja de claves pública / privada y que normalmente se almacenan en una cartera (wallet), que no es más que un fichero utilizado por diferentes clientes Bitcoin para realizar transacciones [12].

Lógicamente, sólo el usuario con su clave privada podrá firmar una transacción para enviar parte de su dinero a otro usuario y de forma análoga al proceso de firma digital, todo el mundo podrá validar esta transacción a partir de la clave pública del emisor. De esta manera, una transacción se puede ver como un mensaje firmado digitalmente que contiene una cantidad de bitcoins a transferir desde un usuario a otro usuario. El hecho de utilizar firmas electrónicas hace que sólo el dueño legítimo de un dinero pueda realizar transacciones con él y sólo el destinatario legítimo de una transacción pueda reclamar su contenido. Cualquier transacción que se realiza es distribuida por la red Bitcoin para que pueda ser validada.

Este proceso plantea un inconveniente fundamental: ¿qué es lo que impide a un usuario enviar la misma cantidad de dinero a dos usuarios diferentes? Esto es lo que se conoce como el doble gasto y es un problema fundamental en cualquier moneda digital. Para poder entender cómo se evita el doble gasto en Bitcoin hay que tener en cuenta las características del protocolo y su funcionamiento:

- Cuando se crea una nueva transacción, todos los detalles de la misma son distribuidas a toda la red.
- Existe una cadena de bloques mantenida por todos los nodos de la red que contiene un registro con todas las transacciones realizadas a lo largo del tiempo. Cada nodo tiene una copia de esta base de datos.
- Todas las transacciones, una vez confirmadas su validez, son incluidas en un bloque.
- Para que un bloque sea aceptado en la cadena de bloques, todas sus transacciones han de ser válidas y además se debe incluir una prueba de trabajo, que es la solución al problema matemático anteriormente mencionado.
- Un bloque se añade a la cadena de manera que si uno es modificado, para que la cadena siga siendo válida, todos los bloques posteriores en la misma tienen que ser recalculados.
- Puede ocurrir un momento en que la cadena se divida en dos ramas totalmente válidas. Si se diera el caso, la se aceptaría como rama principal aquella que sea más larga.

De esta manera, cuando un usuario ve que una transacción que ha realizado se encuentra dentro de un bloque que está en la cadena principal, entonces sabrá que su transacción ha sido confirmada por la red.

Si un usuario quisiera gastar de nuevo este dinero, entonces tendría que construir una nueva cadena en la que la transacción realizada se sustituyera por la nueva y que esta se convierta en la cadena principal. Para realizar esto, es necesario tener más capacidad de cómputo que el resto de la red, pues supondría tener que modificar todos los bloques de la cadena posteriores al bloque que contiene la transacción y, como se ha visto, generar un bloque válido conlleva resolver un problema difícil. Bitcoin se basa en lo que decida la mayoría, por lo que mientras la mayoría de nodos siga las reglas establecidas, se evitará el doble gasto.

A medida que se añaden más y más bloques encima de una transacción, ésta se vuelve más difícil de modificar.

Por tanto, los elementos que componen el sistema Bitcoin se pueden descomponer en los siguientes:

- Transacciones. Una transacción consiste en una transferencia de valor entre dos usuarios (wallets) que será incluida en la cadena de bloques. Un wallet está compuesto por una clave pública y una clave privada que es utilizada para firmar las transacciones, garantizando la autenticidad e integridad de las mismas. Todas las transacciones son distribuidas por la red y son confirmadas mediante el proceso de minado.
- La cadena de bloques. La cadena de bloques consiste en una base de datos distribuida en la que se basa el funcionamiento del protocolo Bitcoin. Todas las transacciones confirmadas se encuentran dentro de algún bloque de la cadena principal. Por medio de la

cadena un usuario puede conocer y calcular en todo momento el balance de sus cuentas, los movimientos y la cantidad de bitcoins que puede gastar. La integridad y cronología de la cadena de bloques se consigue mediante el uso de diferentes primitivas criptográficas.

- Bloques. El minado es un proceso basado en consensos y acuerdos en la red que se utiliza para confirmar las nuevas transacciones e incluirlas en la cadena de bloques. Este proceso protege la neutralidad de la red. Para que un conjunto de transacciones se puedan añadir a la cadena es necesario empaquetarlas en bloques, que como se ha visto, tienen que cumplir una serie de requisitos para poder ser considerados válidos por la red. Estas reglas hace posible que los bloques de la cadena no puedan ser modificados y es el proceso por el que se generan nuevas monedas en el sistema.

2.4 Bitcoin en la práctica

Bitcoin es hoy en día una realidad y como tal, son muchos los usuarios y comercios que la aceptan como moneda para realizar pagos. Se pueden adquirir productos y servicios en toda clase de negocios como restaurantes, hoteles,.. Además se pueden adquirir servicios online en reconocidas compañías como Wordpress, Redis o Flattr. En [11] y [1] se pueden encontrar mapas con los comercios que aceptan Bitcoin a lo largo del mundo. El número de negocios no para de crecer y así lo demuestran los números. Actualmente el valor de mercado de todos los bitcoins en circulación supera los 8 mil millones de dólares con miles de transacciones realizadas diariamente [13].

Hacer una transacción con Bitcoin es un proceso tan sencillo como descargarse una aplicación para el móvil [14], un proceso mucho más simple que el necesario para poder realizar un pago con tarjeta de crédito si no se dispone de una. Lo único que hay que hacer para realizar una transacción es incluir la dirección (clave pública) del destinatario y la cantidad de dinero a transferir. Una clave pública no es fácil de recordar o transmitir como un número de teléfono o un email. Por ello muchas de las carteras hacen uso de tecnologías como códigos QR [15] o NFC [16] para hacer el proceso más sencillo y amigable.

Para poder hacer pagos en Bitcoin primero es necesario adquirir bitcoins. Existen varias maneras de adquirir esta moneda:

- Minado. Las monedas entran en circulación mediante este proceso, por lo que una forma de adquirirlas es colaborando con la red. Sin embargo, a medida que la red ha ido creciendo, la dificultad para generar un bloque ha ido creciendo (con el objetivo de mantener el ratio de 1 bloque cada 10 minutos), por lo que intentar obtener bitcoins usando un simple ordenador sería algo impráctico. Para obtener algún beneficio en el proceso de minado, normalmente es necesario adquirir hardware especializado en el minado de Bitcoin [17], cuyo precio puede variar de cientos a algunos miles de dólares. Obtener bitcoins con este tipo de hardware facilita el proceso, pero todavía sigue siendo complicado, por lo que muchos usuarios deciden asociarse para formar lo que se conocen mining pools [18], compartir su capacidad de cómputo, facilitando el proceso y así repartir los beneficios equitativamente en función de los recursos aportados. Existen pools que alcanzan a tener hoy en día hasta el 38% de la capacidad de cómputo de la red [19].
- Recibirla a cambio de bienes y servicios.
- Adquirir bitcoins a cambio de dinero real. Existen diferentes casas dedicadas a la compraventa de bitcoins. En [20] se puede encontrar un listado por países de los lugares en los que se pueden adquirir. Además, existen herramientas como LocalBItcoins [21] que muestra listados de usuarios interesados en vender / comprar bitcoins en diferentes ciudades del mundo.

2.5 Controversia

A lo largo de su corto tiempo de vida, Bitcoin se ha visto envuelto en una serie de controversias o preocupaciones originadas en sus propias características y los usos que se le

puedan a la moneda. Así a pesar de que ningún país ha declarado Bitcoin ilegal, algunos estados sí que han establecido bastantes limitaciones al uso de la moneda, como por ejemplo en China o en Rusia, donde en enero de 2014 éste último propuso limitar el uso de bitcoin, estableciendo un máximo de unos 30 dólares diarios por persona [22]. El caso más desfavorable ocurre en Vietnam, donde el banco central prohibió el uso de Bitcoin para cualquier tipo de intercambio [23].

Las características de anonimato e irreversibilidad de las transacciones puede llevar a pensar que Bitcoin se encuentra un paso por detrás frente a los sistemas tradicionales en lo que se refiere en la lucha contra el fraude financiero, lavado de dinero y otras actividades ilegales. Existen wallets como Dark Wallet [24] que hacen prácticamente imposible la trazabilidad de una transacción. Sin embargo, esto no es nuevo, ya que el dinero en efectivo ofrece las mismas características y a pesar de ello es ampliamente utilizado.

Por último, en los últimos meses el valor de mercado de Bitcoin crecido de forma extraordinaria, llegando a aumentar su valor de apenas 200 a más de 1000 dólares en apenas unos días. Esta serie de hechos puede hacer pensar que Bitcoin se trata de una burbuja.

3 Ataques a Bitcoin

El sistema Bitcoin, su protocolo y las primitivas criptográficas utilizadas tienen un sólido historial de seguridad y es, posiblemente, una de las mayores redes de computación distribuidas del mundo [25]. Algunos de los fallos y robos que se producen en Bitcoin se deben a comportamientos que tienen su origen en errores de los usuarios. Una cartera (wallet) contiene las claves privadas necesarias para poder enviar y recibir dinero y al igual que una cartera normal, esta puede ser perdida, borrada o robada. Existen una serie de recomendaciones para que los usuarios puedan hacer uso de Bitcoin con mayor confianza y seguridad [26].

Las reglas en las que se basa el protocolo Bitcoin han estado funcionando de la misma manera (salvo pequeñas actualizaciones) desde el momento de su concepción, lo que es un buen indicador que los conceptos fundamentales están bien diseñados. Sin embargo, es cierto que se han descubierto una serie de fallos de seguridad en las implementaciones de algunos de los clientes del sistema [27] y la seguridad de estas implementaciones dependerá de la velocidad con que se corrijan estos errores a medida que son descubiertos. Mientras más errores se descubren y arreglan, más madurez va adquiriendo el protocolo.

Forzar o cambiar el comportamiento del protocolo Bitcoin no es una tarea sencilla. Cualquier nodo que no siga las reglas no puede forzar el uso de sus reglas a otros usuarios. A partir de la especificación actual, mientras todos los nodos sigan las reglas, no será posible realizar un doble gasto en la misma cadena, como tampoco será posible reclamar una serie de bitcoins sin la firma adecuada. No es posible generar una cantidad incontrolada de bitcoins de la nada, usar los fondos de otros usuarios o corromper la red. Es un protocolo que se basa en lo que decida la mayoría.

Sin embargo, una mayoría de nodos (y por mayoría se quiere decir un conjunto o individuo que tenga más del 50% de capacidad de computación de la red) podría decidir bloquear o deshacer transacciones recientes. Al mismo tiempo esta mayoría podría presionar para forzar una serie de cambios.

Para que el funcionamiento de Bitcoin sea correcto, tiene que haber consenso entre todos los usuarios y por tanto, para alterar las reglas, sería necesaria una gran mayoría de usuarios que no dejaran al resto de usuarios más que la alternativa a aceptar el cambio. Sería difícil entender por qué un usuario querría adoptar un cambio que pusiera en peligro su propio dinero.

A pesar de esto, desde la concepción del propio Bitcoin era conocido que un usuario o grupo de usuarios con un poder de más del 50% de la capacidad de cómputo de la red sería capaz de realizar dobles gastos y bloquear transacciones. Además de este ataque, con el paso del tiempo han ido surgiendo otra serie de ataques a nivel teórico que podrían llegar a suponer un peligro para el buen funcionamiento de la red. A lo largo de este capítulo se intentará hacer una descripción de los ataques más importantes que se han planteado contra el protocolo Bitcoin, la red o sus usuarios.

3.1 Ataque del 51%

Sin duda alguna, este ataque es la más y mejor conocida de las vulnerabilidades del protocolo Bitcoin y ha existido desde su diseño, como bien se indica en el artículo original en el que se presentó esta cripto-moneda [3]. Bitcoin almacena toda la información de las transacciones en la cadena de bloques. A medida que se van generando nuevas transacciones, estas serán añadidas a un bloque que finalmente pase a formar parte de la cadena principal. Puede ocurrir que en un momento determinado se produzca una bifurcación en la cadena, siendo solamente la de mayor longitud (dificultad) la considerada válida.

De esta manera un usuario podría intencionadamente crear dos transacciones que entren en conflicto, en la que en una se envía dinero a un comerciante y en la otra se envía el mismo dinero a una cuenta que estuviera en su posesión. El atacante retransmite la primera de las

transacciones al mismo tiempo que intenta construir secretamente una bifurcación en la cadena a partir de la segunda de las transacciones. Después de que la transacción tenga n transacciones, el comerciante enviará su producto. Si llegado este punto el atacante ha sido capaz de generar más de n bloques, podrá liberar esta nueva cadena y recuperar sus monedas al mismo tiempo que conserva el producto que le fue enviado. La probabilidad de éxito de este ataque depende de su capacidad de computación y del número de confirmaciones con las que el comerciante da por válida una transacción. Si ocurre que un atacante controla más de la mitad de la capacidad de cómputo de la red, este ataque tiene una probabilidad de éxito del 100% puesto que el atacante es capaz de generar bloques a un ritmo más rápido que el resto de la red.

A su vez, un atacante que tenga tal capacidad de cómputo será capaz de bloquear algunas o todas las transacciones o que otros nodos que se encuentren minando bloques puedan incluirlos en la cadena principal. Sin embargo, un atacante no podría nunca deshacer las transacciones de otros usuarios, cambiar el número de monedas generadas por bloque, crear monedas de la nada o utilizar bitcoins que nunca que le pertenecieron.

A medida que se va descendiendo en la cadena, la dificultad de modificar estos bloques va creciendo de forma exponencial.

A pesar de que es posible llevar a cabo este ataque (especialmente con la expansión de los mining pools), la cantidad de recursos que requiere en comparación con la cantidad de poder y control que ofrece sobre la red es poco probable que se intente llevar a cabo.

3.2 Race attack

Dado el funcionamiento de Bitcoin y el tiempo necesario para confirmar una transacción (para que sea incluida en un bloque deberán pasar al menos 10 minutos y algo más de tiempo para tener varias confirmaciones), hay ciertos negocios que dada su naturaleza no pueden esperar este tiempo para ofrecer sus productos (restaurantes de comida rápida por ejemplo) y por ello dan un pago por válido cuando detectan que una transacción ha sido enviada a la red para ser incluida en un bloque (0 confirmaciones). De esta manera, un atacante, emulando el comportamiento del ataque anterior, enviará una transacción a toda la red con el dinero a una cuenta que controle y en el momento de efectuar el pago enviará una transacción al comerciante con el mismo dinero, pero esta será al poco tiempo considerada inválida, al ser la anterior incluida en un bloque.

Los comerciantes pueden protegerse de este ataque desactivando las conexiones entrantes y conectándose sólo a determinados nodos conocidos. En [28] se proponen una serie de medidas extra para protegerse y poder detectar estos ataques.

3.3 Ataque de Sybil

Este ataque propio de las redes comunicaciones [29] tiene como objetivo aislar un nodo del resto de la red para bloquear sus transacciones por ejemplo. Para poder llevar a cabo este ataque, un atacante deberá tener bajo su control un número indeterminado de nodos, de tal manera que la víctima sólo esté conectada a estos nodos y poder provocar una desconexión de la red bloqueando sus transacciones y bloques, descubrir la identidad del usuario de las transacciones o el atacante podría llegar a hacer ejecutar un doble gasto de sus bitcoins.

Aunque teóricamente es posible llevarlo a cabo, en la práctica ejecutar este ataque de forma satisfactoria es algo realmente complicado. Focalizar un ataque en un nodo particular es algo complicado. Lo es más si el nodo que se quiere atacar es por ejemplo un comercio (casas de cambio, o algún otro nodo en el que realizar un doble gasto conllevara obtener una gran cantidad de dinero), ya que es más que probable que este nodo tenga establecidas muchas conexiones con nodos legítimos, por lo que sería complicado aislarlo de esos nodos. Además para que el ataque tenga éxito, es necesario proveer a la víctima para que no sospeche durante la duración del ataque una cadena de bloques falsa. Generar esta cadena de bloques es

complicado, a no ser que se invierta dinero en poder de cómputo, algo que carece de sentido, al estar invirtiendo dinero en generar bloques que carecen de valor para llevar a cabo un posible doble gasto de menor valor. Al no poder proveer una cadena realmente falsa, la víctima se comenzará a preguntar por qué sus transacciones tardan más tiempo en confirmarse de lo normal.

3.4 Ataques DoS y escuchas

Un usuario que pueda ver el tráfico de un nodo, fácilmente podrá detectar cuándo un usuario retransmite una transacción que previamente no había recibido, es decir, que es generada por el propio usuario y así adivinar su identidad en la red. Algunos clientes de Bitcoin tienen soporte para mejorar el anonimato mediante el uso de la red Tor [30].

El envío de grandes cantidades de datos a un nodo puede hacer que se quede ocupado y no sea capaz de procesar las transacciones Bitcoin normales. El protocolo en sí mismo y las diferentes implementaciones incluyen una serie de medidas para protegerse contra ataques de denegación de servicio. Sin embargo, es posible que se puedan ejecutar ciertos ataques más sofisticados. De entre las reglas definidas en el protocolo para evitar estos ataques destacan:

1. Limitar el tamaño máximo de un bloque a 1 megabyte.
2. Limitar el número de comprobaciones de firma que una entrada a una transacción puede solicitar mediante el sistema de scripting.
3. Limitar el tamaño de cada script a 10000 bytes.
4. Limitar el tamaño de cada valor que se pueda usar en un script a 520 bytes.
5. Limitar el número de operaciones complejas que se pueden realizar en un script.
6. Limitar el número de firmas que el comando para la verificación múltiple de firmas puede tomar (20 claves).
7. Limitar el número de elementos que se pueden almacenar simultáneamente en la pila de scripting a 1000.
8. Limitar el número de comprobaciones de firma que un bloque puede solicitar a 20000.

3.5 Selfish attack

Uno de los requisitos para que un atacante sea capaz de interferir en la red Bitcoin y poder bloquear y deshacer transacciones es que éste tenga más capacidad de cómputo que el resto de la red junta. Para poder ejecutar este ataque, se requiere disponer lógicamente de más del 50% de la capacidad de cómputo de la red, como se veía anteriormente en el estudio del ataque del 51%. Sin embargo, recientemente se ha publicado un nuevo ataque [31] que reduce de forma significante la premisa anterior.

Para llevarlo a cabo, en lugar de actuar como un nodo normal y difundir los bloques a toda la red a medida que los va descubriendo, un atacante publicará estos bloques de forma selectiva, llegando a sacrificar algunas veces sus propios beneficios pero en la mayoría de las ocasiones publicando una gran cantidad de bloques simultáneamente que hará que el resto de la red descarte algunos bloques y pierda beneficios. Este comportamiento reduce los beneficios que el atacante recibe a corto plazo, pero a largo plazo reduce el beneficio del resto de los nodos todavía más.

Estos nodos neutrales se plantearán la idea de unirse al atacante para aumentar sus propios beneficios. Llegará un punto en que este grupo crecerá hasta ser una mayoría, dando al atacante un alto grado de control sobre la red.

De forma más concreta, un atacante tendrá en su haber una cadena privada, que será totalmente independiente de la cadena pública que el resto de nodos de la red comparte. Al comienzo del ataque estas dos cadenas parten del mismo punto. A partir de aquí el atacante siempre intentará crear nuevos bloques sobre su cadena privada, manteniendo en secreto todos aquellos bloques que descubra. En función de la capacidad de cómputo del atacante y del

porcentaje de nodos de la red que apoya al atacante se puede determinar cuándo hay que hacer pública esta cadena.

Sea X el porcentaje de cómputo del atacante frente a la red y Z el porcentaje de nodos que lo apoya, el ataque puede pasar por diferentes estados:

- Estado 0. Si la cadena privada y la cadena pública son iguales, entonces el atacante ha de trabajar en la cadena privada. Con probabilidad X, el atacante descubrirá un nuevo bloque y pasará al estado 1. Con probabilidad 1 - X la red descubrirá un nuevo nodo y el atacante deberá resetear su cadena privada.
- Estado 1. Si la cadena privada es un bloque más larga que la cadena pública, se debe seguir trabajando en esta cadena. De nuevo, con probabilidad X el atacante descubrirá un nuevo bloque y se moverá al estado 2. Si la red descubre un nodo antes (probabilidad 1 - X), entonces se deberá ir al estado 0'.
- Estado 2. Trabajando de nuevo en la cadena, con probabilidad X el atacante se moverá al estado 3 y con probabilidad 1 - X la red encontrará un bloque y entonces el atacante liberará si cadena, que estará un bloque por encima de la cadena real, por lo que la red utilizará esa cadena y el atacante tendrá un beneficio de 2.
- Estado n (n > 2). Con probabilidad X el atacante se mueve al estado n + 1 y aumenta su valor en 1 cuando libere su cadena privada. Con probabilidad 1 -x el atacante regresará al estado n - 1.
- Estado 0'. El atacante publica su bloque, por lo que en este momento hay dos cadenas con la misma longitud. Con probabilidad X el atacante descubrirá un nuevo bloque y la red tomará su cadena como válida obteniendo un beneficio de 2 y volviendo al estado 0. Por otro lado, con probabilidad (1 - X) Z, la red encuentra un bloque en la cadena del atacante y así el atacante y la red obtienen un beneficio de 1 volviendo al estado 0. Por último, con probabilidad (1-X) (1-Z) la red encontrará un bloque encima de la cadena principal, ganando la red un beneficio de 2 y volviendo al estado 0.

El éxito de este ataque depende de los valores de Z y de X. En el artículo en el que se presenta este ataque se demuestra que con Z = 5 y con un valor de X = ¼ el atacante se vuelve más eficiente que el resto de la red y con un valor de X = ⅓, el atacante se vuelve más eficiente que la red pública independientemente del valor de Z.

En la red Bitcoin actual existen algunos mining pools que tiene esta capacidad de cómputo y podrían decidir empezar este ataque, aunque no parece haberse convertido en un problema para la red dado el soporte altruista que aportan la mayoría de nodos y debido a que estos no quieren desestabilizar una fuente de sus ingresos. En [32] se discute la posibilidad de que los mining pools lleguen a convertirse en un cartel, que es la idea que hay detrás de este ataque.

4 Diseño del simulador Bitcoin

4.1 Estado del arte

Como se ha podido ver en esta memoria, desde su nacimiento en 2009 el uso de Bitcoin ha crecido de forma extraordinaria y son muchas las aplicaciones existentes en torno a la moneda. Sin embargo, la variedad de soluciones o proyectos que ofrezcan programas relacionados con el desarrollado en este trabajo es muy limitada. El trabajo realizado en este ámbito no es tan amplio y no son muchas las referencias que se puedan encontrar al respecto.

De los elementos encontrados, el que más destaca es un simulador escrito en Python [33] que permite simular la generación de bloques en la cadena y la propagación de los bloques a través de la red. Está basado en un sistema dirigido por eventos discretos. Al finalizar la ejecución se muestran al usuario una serie de ventanas con información acerca de la simulación. Es un simulador muy completo que implementa bastantes características del protocolo, pero al que le falta mayor completitud a la hora de poder visualizar los resultados de la simulación.

Otro de los proyectos interesantes es la implementación en Coffeescript de un simulador de red genérico que incluye una biblioteca de simulación del cliente Bitcoin todavía en desarrollo [34]. Por esto, está más orientado a ser un simulador de red de carácter general más que al protocolo Bitcoin en sí mismo. Este simulador se puede ejecutar como un proceso o desde una página web en la que ir viendo en tiempo real el estado de la red y los eventos que se producen a medida que aumenta el tiempo. Si se analiza un poco el código de este simulador, se observa que incluye una gran cantidad elementos y componentes como el descubrimiento de nodos y que además permite el uso de plugins. Dentro del módulo Bitcoin, se llega a emular hasta la generación y validación de las transacciones.

Sin duda alguna, estas implementaciones se convierten en soluciones estupendas para tomar como punto de partida y desarrollar un simulador de mayor alcance capaz de emular en su completitud el protocolo Bitcoin no sólo en sus especificaciones sino en magnitud. Sería ideal contar con la infraestructura para poder realizar simulaciones con gran cantidad de nodos, pudiéndose aproximar de forma más fiel a lo que ocurre en la realidad, donde la red Bitcoin está compuesta por miles de nodos y cuyo tamaño no dejar de crecer. Otro aspecto importante de la simulación es la capacidad de análisis de los eventos y comportamientos acontecidos. Este es un aspecto fundamental, pues si no, el esfuerzo realizado en reproducir en un entorno controlado lo que ocurre en el mundo real carecería de sentido. Quizás este aspecto sea uno de los menos desarrollados en los simuladores introducidos en este apartado.

A lo largo de los próximos capítulos se entrará en los detalles del simulador que se ha sido desarrollado durante el periodo de ejecución de este trabajo y en el que se ha hecho especial hincapié en tres aspectos fundamentales: infraestructura, escalabilidad y post análisis.

4.2 Delimitación del alcance del simulador

Antes de comenzar a diseñar la estructura del simulador, era necesario marcar el alcance y los elementos del protocolo que se pretendían conseguir con el mismo teniendo en cuenta el tiempo del que se disponía para el desarrollo del simulador y también teniendo en cuenta que se quería terminar el proyecto con un producto funcional.

Lenguaje de programación

Uno de los requisitos del simulador era que estuviera escrito en Python, ya que hay mucho desarrollado sobre Bitcoin en este lenguaje y así se facilitaba un posterior desarrollo del simulador una vez terminado este trabajo.

Uno de los requisitos que se establecieron desde el principio para el desarrollo del simulador era que éste, especialmente su núcleo estuvieran escritos en Python. Las razones de esta elección no son aleatorias, sino que hay unas razones de peso que la respaldan.

Existe una gran literatura, bibliotecas y otro tipo de componentes desarrollados en este lenguaje acerca del protocolo y el ecosistema Bitcoin. Cabe destacar la implementación de un cliente Bitcoin en Python [35], de una serie de bibliotecas para la interacción con las estructuras de datos del protocolo [36] y la implementación del simulador introducida en el apartado anterior [33].

Desde un principio siempre existió la duda de si implementar el simulador como una aplicación independiente o dentro de un simulador de red ya existente. Uno de los simuladores de red más extendidos dentro del mundo de la investigación es NS3 [37], el cual permite desarrollar módulos y plugins que estén escritos en Python.

Si se pretende desarrollar un simulador que sea lo más fiel que se pueda a la realidad, sin duda alguna, es necesario utilizar diferentes modelos matemáticos y probabilísticos. En este aspecto, dentro del lenguaje Python destaca la biblioteca NumPy [38], ampliamente utilizada en el ámbito científico para tal fin.

La elección de este lenguaje también suponía un reto, pues no se tenía ningún conocimiento acerca del mismo antes de comenzar el desarrollo del proyecto, lo que aportaba un valor añadido a la elección de este lenguaje, convirtiendo el trabajo en una aventura doblemente dinámica.

Partiendo de cero

Una de las primeras cuestiones que se plantea a la hora de plantear qué es lo que se quería conseguir y hasta dónde se pretendía llegar con el simulador pasaba por decidir entre si hacer un desarrollo desde cero o si por el contrario desarrollar algún tipo de módulo o plugin que se pudiera embeber en un simulador de red ya existente como el ya mencionado NS3.

Ante estas dos alternativas, se optó por realizar un desarrollo propio, pues en términos generales permitiría tener un mayor control sobre el producto final y se evitaría tener que aprender primero a trabajar con el simulador que se eligiera, su arquitectura y modo de funcionamiento para a continuación poder comenzar el diseño y desarrollo del simulador. Quizás esta elección tampoco habría sido desacertada, pero el tiempo era un factor en contra y siempre se tenía en mente la idea de presentar un producto más o menos desarrollado pero funcional y por ello se decidió implementar el simulador partiendo de cero.

Otro de los motivos por los que la balanza se inclinó en esta dirección fue porque el objetivo principal era simular el comportamiento de los nodos de la red Bitcoin sin entrar en los detalles del protocolo de red subyacente, es decir, el protocolo IP. Se pretendía quedarse un nivel por encima, y así se ha podido ver a lo largo de esta memoria, en la que la explicación del protocolo no ha profundizado apenas en aspectos específicos del protocolo de red utilizado. Por esto, el hecho de embeber el desarrollo en un simulador de red genérico iba perdiendo sentido: se estaría sobrecargando la aplicación con elementos y componentes que no serían necesarios o que su uso no aportaría apenas en la consecución de los objetivos perseguidos.

Si se desarrollaba un simulador desde cero, se tendría un mayor control y conocimiento sobre cada uno de los pasos que se iban dando, cumpliendo con uno de los objetivos de cualquier trabajo de final de master como es la adquisición de nuevos conocimientos. Además, se contaba con la existencia de un simulador funcional que se podría tomar como punto de partida y que ayudaría en la toma de decisiones en cuanto al diseño e implementación del nuevo simulador.

Comportamientos a observar

En el capítulo anterior se introdujeron algunos de los ataques más conocidos y que más daño pueden provocar a la red. El éxito de algunos de estos ataques como la escucha de paquetes o las denegaciones de servicio depende directamente de aspectos propios de los protocolos de red subyacentes, algo que se había decidido no implementar en el simulador. Otros ataques como el Race attack basan su éxito en las confirmaciones necesarias para dar una transacción como válida. Este hecho marca como requisito no realizar una implementación de las transacciones, sino también realizar un control y seguimiento de las mismas. Este tipo de ataque es factible en la red Bitcoin, pero afecta a nodos de forma individual y no a la red de forma colectiva.

Si se quería implementar un simulador de Bitcoin para poder observar qué ocurre en la red a partir del comportamiento de los nodos, era necesario comenzar por aquellos ataques que realmente suponían un peligro real a la red, aquellos que fueran capaces de tambalear los cimientos sobre los que se construye el protocolo. Dentro de este tipo de ataques destacan especialmente dos: el ataque del 51% y el selfish attack. Estos ataques, que son los más devastadores que puede sufrir la red Bitcoin explotan aspectos propios del diseño del mismo sin entrar en cuestiones relacionadas con los protocolos de red o las transacciones, sino que basan su éxito en la manera en que se generan bloques y se toman decisiones en la red. El éxito de estos ataques depende fundamentalmente en la capacidad de cómputo del ejecutor frente al resto de la red.

En el ataque del 51% si un nodo es capaz de generar bloques de generar bloques de forma más rápida que la red, será capaz de realizar un doble gasto o de bloquear ciertas transacciones. En el selfish attack, un nodo con una cierta capacidad de cómputo razonable de la red (menor que el 51%), no compartiendo los bloques que descubre de forma instantánea y contando con el respaldo de ciertos nodos, logrará aumentar sus beneficios y posiblemente pueda llegar a controlar la red de igual manera que en el ataque anterior.

Sin duda alguna si se quiere comenzar un simulador de Bitcoin, los primeros comportamientos a observar tiene que pasar por estos dos ataques. Por tanto, se marcó como objetivo realizar una implementación de estos ataques en el simulador al tratarse los dos más importantes de los estudiados en el capítulo anterior.

Bloques, minado, y difusión

Los dos ataques que se habían propuesto implementar están directamente relacionados con el proceso de minado y la capacidad de cómputo de los nodos de la red, sin tener en cuenta aspectos de más bajo nivel como son las transacciones y su proceso de generación o difusión.

Por tanto, a partir de este objetivo, se podía deducir qué aspectos del protocolo Bitcoin era estrictamente necesario implementar. El proceso de generación de bloques, su retransmisión y su inclusión en la cadena principal sería suficiente para implementar estos dos ataques y se podrían obtener resultados bastante positivos y su implementación sería posible en un tiempo razonable, dejando atrás procesos más complicados que aportaría poco desde el punto de vista de estos ataques al protocolo. Así, se decidió no implementar el proceso de creación y verificación de transacciones, ahorrando tiempo de desarrollo sin sacrificar demasiado el trabajo final.

Es cierto que con esta decisión se perdía la posibilidad de simular aquellos ataques que dependían de la validación de las transacciones como ya se ha comentado, pero se ganaba mucho tiempo que se podría dedicar al desarrollo de otros componentes también necesarios e importantes dentro del proceso de la simulación

De esta manera, se podría dar por supuesto que un bloque sería una caja negra que contendría una serie de transacciones (indiferentes para el simulador). Se evitaba el proceso de tener que generar de forma aleatoria transacciones entre nodos para poder difundirlas, validarlas y añadirlas a un bloque que posteriormente fuera generado. El objetivo era emular

el proceso de generación de un bloque y no de su contenido. Con esta decisión, también se descarta la generación de nuevos bitcoins al no haber transacciones en los que invertirlos y las recompensas por el minado de los bloques.

Por tanto, se puede concluir que un bloque se encargaría de generar bloques que tendrían un contenido y tamaño aleatorio. La cuestión era cómo simular el proceso de minado de un bloque. Este es un proceso que depende directamente de la capacidad de cómputo de un nodo, y esta puede ser vista como un porcentaje del total de la capacidad de cómputo de la red. Utilizando este porcentaje y números aleatorios se puede estimar el tiempo en que un nodo dará con un bloque válido. Es más, se puede conseguir de forma sencilla que el ritmo de generación de los bloques se aproxime al valor establecido de 1 bloque cada 10 minutos.

Creación y descubrimiento de nodos

En la red Bitcoin el número de nodos de la red no es invariable, sino que este va cambiando dinámicamente y de forma impredecible. Además de variar el número de nodos también puede variar la capacidad de cómputo global de la red, haciendo que sea necesario ajustar el parámetro de la dificultad de la red para mantener el ritmo de generación de bloques más o menos constante. Además existen diferentes mecanismos para el descubrimiento de nuevos nodos en la red así como la gestión de las conexiones entre estos.

Si se pretendiera hacer variable el número de nodos de una simulación durante su ejecución, esto plantearía una serie de problemas a resolver en lo que se refiere al proceso de minado. Si este número se mantiene invariable, al igual que la capacidad de cómputo de la red, el proceso de minado es sencillo de emular.

Permitir cambios dinámicos en el número de nodos, su capacidad de cómputo o de los otros nodos que conocen supondría tener que implementar mecanismos de descubrimiento y el modelo probabilístico establecido para la generación de los bloques se complicaría.

De nuevo, esta implementación enriquecería el simulador pero no sería de gran relevancia para los ataques que se pretendían ejecutar, por lo que se descartó la creación de nuevos nodos o enlaces entre ellos durante la simulación.

Así, el número de nodos de la red y de las conexiones entre ellos sería prestablecido antes del comienzo de una simulación. Estas interconexiones se generan de tal forma que la que la probabilidad de que exista un enlace entre dos nodos es constante para todos los enlaces y a su vez independiente de los demás enlaces siguiendo un modelo tipo Erdos-Renyi [39]. A su vez, y con intención de aproximar el proceso de transmisión de bloques a un entorno real de una red, estos enlaces tendrían un ancho de banda y una latencia aleatorios y diferente para cada uno de ellos que permitiría emular de forma sencilla todos aquellos procesos propios de la capa de red (IP) que se habían descartado desde un principio.

El protocolo Bitcoin

Con todo esto ya se tenía totalmente clara la estructura de la red que se implementaría dentro del simulador, las interconexiones entre los nodos y la forma en que los nodos trabajarían para generar nuevos bloques que añadir a la cadena principal. Se tendría una red con un número preestablecido de clientes Bitcoin interconectados entre sí de forma aleatoria mediante una serie de enlaces con ancho de banda y latencia aleatoria. Además, cada uno de estos nodos tiene asignado un porcentaje del total de la capacidad de cómputo de la red, por lo que iría generando bloques en función de este y a un ritmo global constante.

El único proceso que queda por definir para tener definido el comportamiento del simulador es la forma en que interactúan los nodos entre sí. Este proceso está bastante claro. Si se quiere simular el protocolo Bitcoin, es lógico que estos actúen de la manera que éste establece en sus especificaciones.

Cuando un nodo descubra un nuevo bloque lo retransmitirá a toda la red (los nodos de los que tenga conocimiento) para que los demás nodos los puedan procesar. Cuando un nodo reciba un bloque que no tiene en su haber, necesitará solicitarlo a los nodos que conoce para poder comprobar su validez y añadirlo a su copia de la cadena principal. Un bloque será válido si su bloque predecesor se encuentra también en la cadena (al no haber transacciones no es necesario realizar comprobaciones de doble gasto por ejemplo). Puede ocurrir que no se tenga la información de un bloque que sea necesario para comprobar la validez de otro que se acaba de recibir. Si ese fuera el caso, el nodo que está realizando el proceso de validación lo solicitará a la red.

Con los objetivos, requisitos y limitaciones que se han establecido, el protocolo que se ha diseñado y que sigue el comportamiento del protocolo Bitcoin resulta bastante sencillo, simple y elegante.

Visualizaciones de datos

Desde el comienzo del trabajo se tenía bastante claro que era fundamental disponer de un módulo de visualización de los datos de la simulación para poder determinar si el comportamiento ocurrido coincide o difiere con el esperado. Es fundamental almacenar de alguna manera todos los datos, interacciones, eventos o cualquier tipo de comunicación que se produzca durante la ejecución de la simulación para su posterior análisis.

La estructura de la red y las conexiones entre los nodos, sus capacidades de cómputo, los bloques que genera y recibe en cada momento, los eventos de red que emite y recibe cada nodo y su estado final han de ser almacenadas.

Con toda esta información sería posible construir un mapa de todo lo ocurrido y éste fue el último requisito que se impuso al simulador que se iba a desarrollar: aparte de almacenar la información, sería necesario crear algún tipo de interfaz que permitiera construir algunas gráficas con los datos generados por cada una de las simulaciones.

Así se puede construir un mapa de todo lo ocurrido y hacerse una idea del estado global de la red.

4.3 Arquitectura

Una vez se tenía claro qué es lo que había que simular era necesario identificar todos y cada uno de los componentes de la aplicación. Sus elementos principales y sus interconexiones. A partir de todas las conclusiones que se habían ido sacando de la fase anterior, era fácil identificar aquellos componentes.

Comenzando con bocetos en papel de las conclusiones obtenidas en la fase anterior, se pudo construir un modelo mental de lo que sería el futuro simulador, pudiendo poco a poco ir deduciendo su estructura y los elementos que debería tener cada componente. La figura 1 representa uno de los primeros bocetos que se dibujaron del simulador, en el que se pueden apreciar algunos de sus componentes como los nodos, los bloques o las interconexiones que conforman la red Bitcoin.

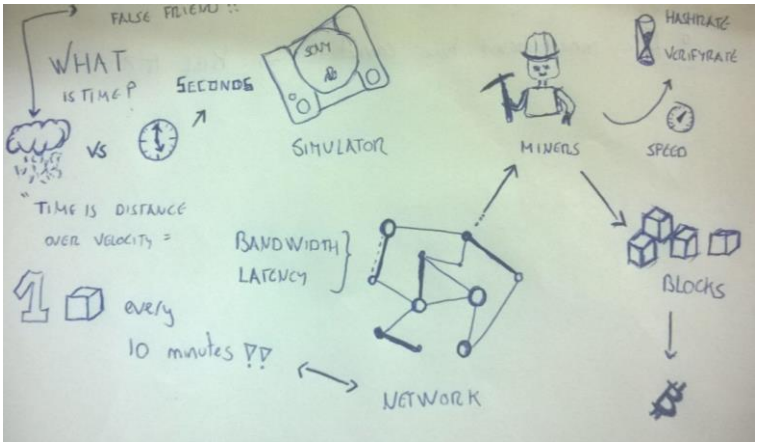

Fig. 1 Boceto de la arquitectura del simulador

Componentes

A partir de los bocetos realizados y del alcance que tendría el simulador se determinó que en el simulador habría tres entidades principales: los nodos, los bloques y los enlaces entre los nodos, siendo los nodos los que tienen toda la lógica de la aplicación, convirtiendo a los bloques y enlaces meros contenedores de información.

Los enlaces juegan un papel fundamental en el simulador, pues son ellos los que definen la estructura de red y qué nodos tienen conocimiento de qué nodos. Además son los encargados de aportar realismo en lo que a las transacciones dentro de la red se refiere al tener cada uno de estos enlaces un ancho de banda y una latencia, haciendo que la información que se transmita entre los nodos no sea instantánea, sino que dependa de su tamaño y del estado de congestión de la red.

Un bloque será la estructura de datos básica y fundamental de la aplicación. De forma análoga al protocolo Bitcoin, cada bloque tendrá un enlace al bloque anterior de la cadena, un identificador del nodo que lo generó, un flag que indica su validez, un tamaño, un parámetro de altura y el tiempo en que fue generado.

- El flag de validez siempre será válido al no existir transacciones en el simulador y al depender la validez de un bloque de su contenido, que será inexistente.
- El tamaño del bloque es la manera en que se consigue simular las transacciones que contiene, que serán consideradas válidas en todo momento. Este campo será generado de forma aleatoria e influirá en el tiempo en que se tarda en enviar a través de la red y en el tiempo en que un nodo tarda en procesar un bloque, ya que en la realidad, como se ha visto, es necesario llevar a cabo un conjunto de comprobaciones que requieren cierto tiempo.
- El parámetro de altura es la manera con la que se consigue simular la dificultad de un bloque y en definitiva es lo que permitirá poder determinar qué cadena es la más larga. Como no se está simulando la dificultad en la generación de los bloques, sino que esta es simplificada aprovechando el hecho de que el número de nodos de la red y su capacidad de computación no varía. Un bloque sólo se añadirá a la cadena principal si su altura es mayor una unidad que su bloque predecesor. Con este parámetro se consigue emular en cierta medida el parámetro de dificultad de forma sencilla (no para la generación de un bloque, pero sí en el momento de añadirlo a la cadena principal). Así, se podrá comparar la dificultad de un bloque frente a otro y poder decidir cuál de ellos tiene prioridad a la hora de añadirlo a la cadena principal.

Por último, un nodo estará intentando generar un nuevo bloque en todo momento a no ser que sea interrumpido por la recepción de un evento a través de uno de los enlaces por los que

está conectado con otros nodos. Este comportamiento no es el seguido por todos los nodos de la red, sino sólo por aquellos que se dediquen a la generación de nuevos bloques. Podrían existir otros nodos que no generen bloques en el simulador, pero al no simularse las transacciones, este comportamiento perdería un poco de sentido.

Cuando un nodo reciba un evento de otro nodo, interrumpirá su proceso de minado y pasará a atender y procesar la información recibida desde la red. El procedimiento para procesar estos eventos llevarán al nodo cierto tiempo en función del tamaño y tipo de evento. Una vez procesado el evento, entonces el nodo puede continuar con el minado de nuevos bloques.

El proceso de descubrir un nuevo bloque llevará al nodo cierto tiempo. Como ya se indicaba, el tiempo que tardará en dar con un bloque válido se determina de forma aleatoria en función de su capacidad de cómputo y el ratio de generación de bloques. Si un nodo es interrumpido mientras está en busca de un nuevo bloque, al retomar el proceso, el tiempo para descubrir un bloque será lógicamente calculado de nuevo. En el momento en el que se descubre un nuevo bloque, se le asigna un tamaño aleatorio, que como se indicaba será determinante en el tiempo que llevará a otro nodo procesar este nuevo bloque.

Puesto que cuando un nodo descubre un nuevo bloque, éste lo difunde a toda la red a través de los enlaces que conoce, también es lógico que reciba bloques que han sido descubiertos por otros nodos. Cuando un nodo recibe un nuevo bloque o lo descubre el propio nodo, deberá procesarlo junto con los bloques que ya tiene y para ello deberá para el proceso de minado.

Los bloques que un nodo tiene en su posesión están clasificados o divididos en tres pools diferentes. Por un lado están aquellos bloques que ya tiene, que son considerados válidos y que pertenecen o han pertenecido a la cadena principal en algún momento (bloques que formaron parte de un fork de la cadena por ejemplo). En segundo lugar están los bloques que ha recibido de otros nodos pero que todavía no ha sido capaz de procesar y por último se encuentran aquellos bloques para los que necesita más información para poder procesarlos (no tiene información de su predecesor salvo su hash) y para los que ha solicitado a la red dicha información.

El procesado de bloques implica recorrer todos los bloques que se encuentre sin procesar, comprobar su validez y, si son válidos se añadirán al pool de bloques válidos. Podría ocurrir que al procesar un bloque, este dé lugar a una cadena mayor que la actual. Entonces, si esto fuera así, este bloque se convertirá en la cabeza de la cadena de esto nodo y éste procederá a notificarlo al resto de la red para que los demás nodos puedan actualizar su cadena si tuvieran que hacerlo. Puede ocurrir que los nodos ya tengan conocimiento de un bloque que reciben. En ese caso no procesarán el bloque, pues ya habrá sido procesado previamente o se encuentra en la cola para ser procesado.

Cuando un nodo está procesando un bloque para el que no dispone toda la información necesaria para poderlo validar, deberá solicitarla al resto de la red y posponer el procesado de este bloque para cuando haya recibido la información que posibilitaría su validación. Este bloque será añadido al pool de bloques pendientes de validación.

Para poder acceder y recorrer la cadena de bloques en todo momento, cada nodo mantendrá una variable en memoria con el hash del bloque que ocupa esa posición en todo momento. Esto evitará tener que hacer números cálculos cada vez que se procese un bloque nuevo y al mismo tiempo permitirá recorrer la cadena de bloques hasta su inicio de forma sencilla.

A pesar de no haber bitcoins ni transacciones, sí que es necesario hacer uso de un bloque génesis a partir del que los nodos comenzarán a generar nuevos bloques. Para identificarlo, este bloque tendrá una altura cero y el identificador del nodo que lo generó será nulo.

Tipos de mensajes y eventos

Dado que un nodo estará constantemente a la espera de recibir eventos e información desde los nodos con los que está conectado, es necesario definir qué tipo de información podrá

recibir de cada uno de estos nodos y el comportamiento que debe de tener ante cada uno de estos eventos, que deberá ser totalmente determinista. Asimismo, se debe de definir qué información y con qué formato los nodos enviarán esta información.

En base al comportamiento de un nodo explicado en el apartado anterior, se ha estimado que este puede ser llevado a cabo con 3 eventos diferentes de red. Estos eventos son los siguientes:

- HEAD_NEW. Este evento es emitido cuando un nodo genera un nuevo bloque válido. Este evento será difundido a través de todos los eventos de la red. Este evento lleva como carga el hash del nuevo bloque descubierto. Cuando un nodo reciba este evento, sólo conocerá el hash de este bloque, por lo que necesitará solicitar al nodo del que recibió el evento toda la información del bloque.
- BLOCK_REQUEST. Este evento se emite cuando se desee obtener la información específica de un bloque. Cuando se emite este evento se envía con él el hash del bloque que se está solicitando. Este evento se puede generar como respuesta a un HEAD_NEW, en cuyo caso sólo se enviará al nodo del que se recibió el evento. Además, este evento se puede también enviar cuando en el momento de procesar un bloque, no se tiene la información completa para verificarlo. En este caso, el evento será emitido a todos los nodos de la red. Cuando un nodo recibe este evento, mirará en su inventario si dispone del bloque solicitado. Si es así, lo devolverá al nodo que lo solicitó. En caso contrario, ignorará el evento y continuará su actividad normal.
- BLOCK_RESPONSE. Un nodo emite este evento únicamente como respuesta a un evento BLOCK_REQUEST. Cuando un nodo recibe este evento, lo que hará será añadirlo a su lista de bloques pendientes de procesar y procederá a procesar todos los bloques que se encuentren en esta lista siguiendo el proceso descrito anteriormente.

Con este sistema de eventos se consigue simular los mensajes de red de inventario que se usaban en el protocolo Bitcoin, mediante los que un nodo informaba con una lista de hashes de los bloques (o transacciones) disponibles y los otros nodos solicitaban información específica de aquellos bloques que le interesaban. Por simplicidad se decidió limitar el tamaño de las listas a un bloque, haciendo que la implementación del proceso sea más sencilla, pero provocando un mayor volumen de eventos en la red.

Mediante la definición de estos tres componentes fundamentales se consigue emular todo el comportamiento del protocolo Bitcoin que se había marcado como objetivo, es decir, el proceso de generación, transmisión, verificación y adición de bloques a la cadena principal. En la figura 2 se aprecian los primeros bocetos de los componentes fundamentales del simulador.

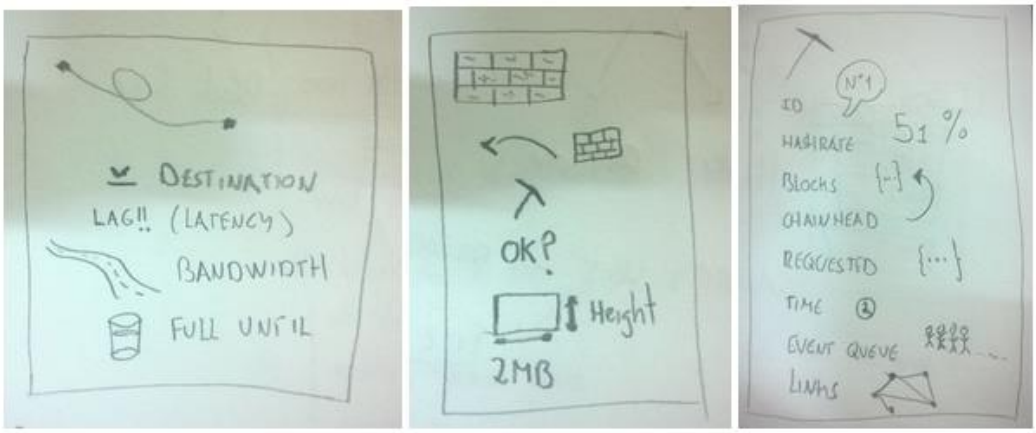

Fig. 2 Boceto de la estructura de un enlace, un bloque y un nodo.

Gestión del tiempo en la simulación

Una de las características del protocolo Bitcoin es que el proceso de minado lleva bastante tiempo. Hay que tener en cuenta que se genera un bloque cada 10 minutos aproximadamente. Por tanto, uno de los aspectos más importantes de la simulación a tener en cuenta es el tiempo y su gestión. Sin duda, juega un papel muy importante en la simulación. Es necesario poder simular el paso del tiempo, pues si esto no es así y se mantienen los tiempos de ejecución reales del protocolo, la ejecución de tal simulación sería inviable e ineficiente desde todos los puntos de vista. Se tiene que poder simular el paso del tiempo, convirtiendo días en milisegundos, haciendo que la simulación de largos periodos se pueda llevar a cabo en instantes.

No sólo es importante lograr la simulación del paso del tiempo, sino que también hay que hacer formar parte de este proceso de emulación todo el proceso de minado y procesado de bloques o el envío y recepción de eventos a través de la red para que pueda existir un orden y una sincronía de manera que todos estos procesos y componentes se ejecuten en el orden, tiempo y forma que se espera. Por tanto, la simulación del paso del tiempo es un elemento tan clave como el resto de los componentes del simulador. Sin ella no se podría avanzar ni obtener resultados.

Ante este problema, se podría optar por implementar un loop de eventos desde cero basado en heaps de eventos (ordenados por el tiempo en que deberían lanzarse) y tener un contador global que simbolizaría el tiempo y que se incrementaría en una unidad después de cada iteración en la que se seleccionará el próximo evento y se procesará. Este procedimiento seguiría así hasta que no hubiera más eventos que procesar o se alcanzara un tiempo límite. Este es el mecanismo empleado por el simulador que se tomó como referencia para el diseño del simulador desarrollado.

Sin duda esta opción destaca por su simplicidad y resultó bastante interesante y atractiva al comienzo por esta razón. Sin embargo, si se tenía en mente los conceptos de arquitectura y escalabilidad que se habían establecido como filosofía del desarrollo, esta pieza no encajaba en ese puzle. Además. Existen distintas implementaciones de simulación de tiempo más ricas, con una API a más alto nivel, y que satisfacían las necesidades del simulador, ¿por qué reinventar la rueda?

De entre las opciones disponibles para construir el simulador, se seleccionó Simpy [40], un framework de simulación de eventos discretos escrito en Python que permite la simulación de redes asíncronas o sistemas multi-agente. Además de proveer clases y elementos para modelar componentes activos (los nodos), provee varios tipos de recursos compartidos que permiten modelar congestión o límites de capacidad (enlaces entre nodos). Por su simplicidad y elegancia se decidió hacer uso de esta biblioteca, que como se verá facilitó mucho el trabajo, aunque bugs contenidos en la misma provocaron algunos quebraderos de cabeza para simular la recepción de eventos a través de todos los enlaces de un nodo.

Monitorización de la simulación

Ya se tenía el qué y el cómo. El único problema que había es que hasta ahora si se lanzaba una simulación, el proceso funcionaría, los nodos se comunicarían entre sí y acabarían generando bloques y añadiéndolos a una cadena que todos compartirían. Sin embargo, no quedaría ninguna evidencia de todo lo acontecido para su posterior análisis. Hacía falta una unidad de persistencia de datos en la que registrar absolutamente todo lo que ocurra durante la simulación.

Se requiere velocidad e independencia del proceso de simulación, es decir, que si el proceso de simulación termina, los datos no se pierdan (no al menos temporalmente) y a su vez que las inserciones en este motor de datos sean lo más rápidas posibles, es decir, que no se hagan escrituras a disco que ralenticen el tiempo total de ejecución de la simulación. Estos dos

requisitos los cumple el almacén Redis [41], un almacén de pares clave-valor en memoria con una velocidad extremadamente rápida para realizar inserciones. No aporta la funcionalidad que pueda aportar un motor de base de datos relacional, pero tampoco es necesario. Utilizando las estructuras de datos proveídas por Redis y un sistema de nomenclatura de claves lógico, se consigue almacenar toda la información que sea generada por el simulador sin aportar grandes retrasos al proceso de simulación.

Esta base de datos contendrá toda la información que se genera durante la simulación, es decir, la información de los nodos y su estado final, sus interconexiones, todos los bloques que se hayan generado (pertenezcan o no a la cadena principal) y todos los eventos que se comuniquen a través de la red.

Con la separación de los datos de la aplicación del propio motor de la aplicación, se podría lograr comparar los datos entre distintas simulaciones llevadas a cabo. Además, si se tiene en cuenta la escalabilidad, esta arquitectura permitiría llevar a cabo simulaciones mucho más poderosas en cuanto a tiempo de ejecución y nodos se refiere, relegando la gestión de los datos a otra entidad y permitiendo que el motor sólo se dedique gestionar las comunicaciones entre los nodos.

Visualización de resultados

Por último, quedaba por plantear la parte que daba sentido a la simulación que es el contenedor en el que poder ver los resultados de la ejecución con distintas visualizaciones que permitan al usuario obtener conclusiones de la ejecución del proceso. La ejecución de una simulación sin obtener resultados de los procesos llevados a cabo carece de total sentido. De entre las opciones posibles para implementar este tipo de aplicación, la alternativa más interesante es sobre una aplicación web puesto que sólo requieren de un navegador para su funcionamiento, sin necesidad de instalación o de estar en la misma máquina en la que se ejecutan las simulaciones para poder acceder a los resultados. De nuevo, esto iría a favor de la escalabilidad, permitiendo múltiples conexiones simultáneas de diferentes usuarios para realizar un análisis de los datos.

Una aplicación web será simple, universal e interactiva. Además de estas ventajas obvias, el hecho de que la web y especialmente JavaScript sean el lenguaje de moda [42], se trata de un entorno bastante conocido, del que se poseen bastantes conocimientos y para el que hay una gran comunidad con un gran soporte y un número inmenso de bibliotecas que facilitan el trabajo. Todos estos hechos harían que el desarrollo de la web para la visualización de los resultados fuera bastante rápido, a diferencia del tiempo planeado para el desarrollo del simulador, ya que Python es un lenguaje totalmente nuevo en el que no se tiene mayor experiencia que la obtenida con el desarrollo de este trabajo.

Esta aplicación web estaría íntegramente programada en JavaScript sin necesidad de la existencia de un servidor para funcionar (webapp HTML5). Este hecho hará que las transiciones entre páginas sean más rápidas y fluidas, conduciendo a mayor usabilidad y user friendliness.

Integración de componentes

Hasta ahora, tras terminar la ejecución de una simulación se tiene una base de datos con todos los datos de la simulación y una aplicación web capaz de dibujar gráficas con estos datos, pero, ¿cómo consigue acceder la aplicación a los datos?

Para responder a este interrogante falta la última pieza del puzle que permitirá interconectar ambos componentes. Se requiere la existencia de un servidor web que sea capaz de proveer a través de una API todos los datos almacenados en Redis a esta aplicación web que mostrará los datos al usuario. Se requiere una API de la que la aplicación web se nutra. Esta API será una API REST que devolverá los datos de la simulación. Puesto que todo el simulador sería

escrito en Python, es lógico seguir esta tendencia y desarrollar el servidor haciendo uso de algún framework también en Python.

Con esto se tendría la arquitectura del simulador y los componentes necesarios para satisfacer los objetivos que se marcaron al comienzo del trabajo. Sin embargo, desde el punto de vista de la usabilidad hay un aspecto que no termina de cuajar en todo este diseño. A pesar de estar todos los componentes interconectados entre ellos, cada uno de ellos es un proceso independiente en el sentido de que para ejecutar una simulación habrá que lanzar el proceso del simulador de forma independiente, que introducirá los datos en Redis y entonces se podrá lanzar el servidor web desde el que accederá a la webapp que usará la API REST para mostrar los datos. No es un proceso natural para el usuario final.

Si se invirtiera este proceso, se conseguiría ganar en usabilidad: se lanza el servidor web desde el que se presenta una aplicación web que permitirá al usuario configurar la simulación y lanzarla. Como la simulación puede tardar tiempo en ejecutarse, será necesario lanzarla en un proceso en segundo plano. Cuando termine la ejecución de la simulación será necesario notificar al usuario e inmediatamente mostrar las diferentes visualizaciones implementadas al usuario. Para poder realizar esto, es necesario utilizar diferentes sistemas de paso de mensajes. Haciendo uso de websockets [43] y del sistema de paso de mensajes de Redis se puede lograr este comportamiento que repercutiría directamente en la usabilidad del simulador.

En los siguientes capítulos se irá desglosando la estructura y el desarrollo de cada uno de estos componentes del trabajo. En la figura 3 se pueden observar todos estos elementos y las relaciones entre ellos:

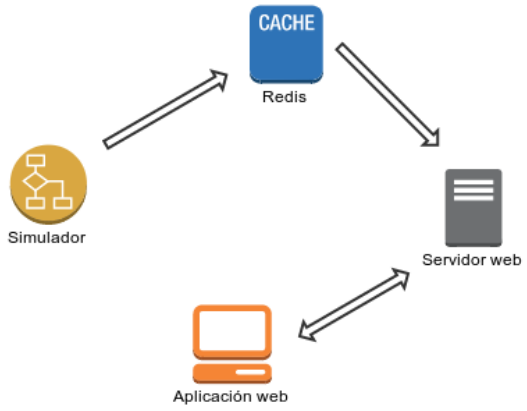

Fig. 3 Relaciones entre los componentes de la aplicación

5 Desarrollo del simulador

En este capítulo se detallaran todos aquellos aspectos que han intervenido en el desarrollo del simulador. Se comenzará con un análisis de las herramientas y tecnologías utilizadas que han posibilitado obtener el software aquí presentado. A continuación se profundizará en la estructura del simulador y cómo el protocolo dibujado a lo largo del capítulo anterior se ha transformado en código. Se analizará la estructura del simulador y se presentarán los principales problemas e inconvenientes que se tuvieron que afrontar. Acto seguido, se introducirá la forma en que se almacenan los datos en Redis, el sistema de persistencia utilizado en la aplicación. Merece la pena pararse en este punto al ser la estructura de esta base de datos totalmente diferente a la de los sistemas gestores de bases de datos relacionales. Este capítulo concluirá presentando algunos datos relativos a la ejecución del simulador, como tiempos de ejecución y consumo de recursos.

5.1 Tecnologías utilizadas

Además de los componentes que se han indicado en el capítulo anterior acerca del diseño del simulador, para el desarrollo de la aplicación se han utilizado una serie de componentes que sin ellas no habría sido posible alcanzar los resultados obtenidos. Por eso, y para poder entender el código de la aplicación desarrollada, es totalmente necesario hacer una presentación de todos aquellos que directa o indirectamente han sido fundamentales para el correcto desarrollo de este trabajo. A continuación se mencionarán las herramientas que más destacan entre las utilizadas.

Pycharm

A la hora de realizar un desarrollo de cierto calibre, no sólo es necesario conocer el lenguaje y todos aquellos elementos que integran el desarrollo. Si se pretende avanzar a un ritmo medianamente rápido, es necesario rodearse de un conjunto de herramientas que automaticen muchos los procesos necesarios y faciliten el proceso de codificación. Existen diferentes IDEs para el desarrollo de aplicaciones en Python. De entre todos los existentes, se eligió el IDE Pycharm [44]. Se trata de conjunto de herramientas que convierten el proceso de programar en una tarea más amigable para el desarrollador aportando sugerencias de código, detección y corrección de errores al vuelo refactorizaciones. Se forma una simbiosis que permite desarrollar un código más elegante y mantenible. La herramienta se encarga de hacer los trabajos rutinarios y el programador se puede centrar en las cosas que aportan valor. La elección de este en particular y no otro es debido a la experiencia que se tiene trabajando con otros IDEs de la misma compañía y que presentan muchas similitudes con Pycharm. Sin duda alguna, ha facilitado procesos como la configuración, gestión de dependencias y depuración del proyecto.

Git

En todo proyecto que se precie es totalmente obligatorio hacer uso de un sistema de control de versiones. Es algo totalmente necesario y no se puede llevar a cabo un proyecto sin tener un control de todos los cambios que se van haciendo en el código. La resolución de bugs o implementación de nuevos features se pueden llevar a cabo de forma seguro manteniendo una referencia en todo momento a aquello que es estable, facilitando por ejemplo la liberación de nuevas versiones. Estas herramientas se vuelven todavía más importantes cuando se trabaja en equipo y son muchos los desarrolladores que están tocando prácticamente el mismo código de forma simultánea. Se pueden detectar los posibles conflictos al mezclar diferentes ramas de código y solucionarlos de forma amigable.

Para este proyecto se hizo uso de Git [45], uno de los CVS (Control Version System) más extendidos en el mundo. La elección de este sistema y no otro, se basa en su popularidad,

extensión y los conocimientos que se tienen acerca del mismo. Un ejemplo de esto es el éxito de plataformas como Github, en donde se encuentra almacenado todo el código desarrollado en este trabajo [46].

Simpy

A lo largo del capítulo anterior ya se comenzó a hablar de esta biblioteca sobre la que se estructura todo el motor del simulador. Es la encargada de la gestión del tiempo en las simulaciones, la que marca el ritmo, orden y tiempo de ejecución de los distintos elementos que dependan de esta magnitud como es el minado y procesado de bloques y el envío de información a través de la red. Además esta biblioteca se utiliza también para la gestión de los diferentes eventos que un nodo estará escuchando. Será el encargado de interrumpir el proceso de minado de un nodo en el momento en que se reciba un evento de red.

Esta biblioteca se trata de un framework de simulación de eventos discretos escrita en Python basada en el uso de Generators [47] para la implementación del despachador de eventos. Sus características la hacen ideal para la simulación de redes asíncronas o de sistemas multi agente.

Se basa en la definición de procesos mediante el uso de Generator que son utilizados para modelar distintas entidades. Además, incluye una serie de utilidades que permiten la creación de distintos tipos de recursos compartidos que permiten modelar puntos de congestión o en el caso del simulador han permitido modelar las conexiones entre los nodos.

Para poder entender el funcionamiento del simulador, es necesario hacer una introducción previa de los componentes fundamentales de Simpy, que son los procesos.

En toda simulación existirán una serie de componentes activos (los nodos) y su comportamiento se modela mediante los procesos. Estos procesos se encuentran dentro de un entorno controlado de simulación e interactúan unos con otros mediante el uso de eventos.

Los procesos se describen mediante Generators, y durante su tiempo de vida irán lanzando diferentes eventos y mediante la sentencia yield. El uso de Estos Generators es lo que permite que un proceso se pueda quedar a la espera sin hacer nada hasta que un evento sea liberado. Por ejemplo, se sabe que un nodo tarda un tiempo aleatorio en dar con un nuevo bloque. Usando uno de los eventos más importantes de Simpy, conocido como Timeout, que serán disparados después de una cierta cantidad de tiempo (simulado), permitiendo que el proceso se duerma o mantenga su estado (en el caso de un nodo, éste seguiría minando) hasta que transcurra ese tiempo.

Los procesos pueden interactuar los unos con los otros. Lo normal es que en una simulación con diferentes agentes activos, éstos se comuniquen entre ellos o que un proceso tenga que esperar la terminación de una acción de otro proceso para poder llevar a cabo sus acciones. Al mismo tiempo un proceso que está esperando por un evento determinado podría ser interrumpido por otro proceso (en el simulador, un nodo dejará de esperar a que pase el tiempo del evento Timeout de minado cuando otro proceso lo interrumpa porque ha recibido un mensaje a través de la red).

Los recursos compartidos que ofrece Simpy, en los que múltiples procesos querrán utilizarlos para poner en común información, o hacer uso de esos recursos compartidos. En el caso del simulador, se hizo uso de un recurso compartido para simular el comportamiento de los enlaces en la red.

Con esta pequeña introducción a los conceptos básicos de Simpy, se podrá comprender mejor la estructura del código cuando esta sea referenciada en los próximos apartados.

Redis

La base de datos Redis es esencial no en el funcionamiento del simulador pero sí en el posterior análisis de los datos e información que éste genera. Es la que permite persistir todo

estos datos de forma extremadamente rápida para que puedan ser consultados con posteridad a la ejecución de la misma y poder obtener conclusiones al respecto de todos los elementos que han tenido lugar durante el proceso de simulación.

Puesto que la estructura de Redis difiere bastante de las bases de datos tradicionales, basadas en el uso de tablas, es necesario hacer una introducción a las principales estructuras de datos y comandos que son ofrecidos por Redis para interactuar con la información.

Redis se trata de un almacén de parejas clave-valor, aunque normalmente se referencia como un servidor de estructuras de datos. Esto es debido a que tiene soporte para cinco estructuras de datos diferentes: cadena, hashes, listas, conjuntos y conjuntos ordenados. Cada uno de ellos tiene una serie de características y comandos únicos. Independientemente del tipo de dato, todos ellos son accesibles mediante una clave, que aunque está formada por un byte de arrays, normalmente se utilizan cadenas de texto como claves.

Redis se caracteriza por ser extremadamente fácil de instalar y configurar. Los datos se almacenan en disco en un único fichero, lo que permite realizar copias de seguridad de forma sencilla. Uno de los requisitos de los datos almacenados en Redis es que estos deben caber en la memoria RAM del equipo, pues se busca optimizar la velocidad de acceso a los datos a través de una API que como se verá a continuación es extremadamente sencilla.

Para terminar este apartado, se introducirán las cinco estructuras de datos soportadas por esta base de datos y que permitirán entender de forma más sencilla la forma en que se almacenan los datos de la simulación.

- Cadenas. Esta es la estructura de datos más sencilla de todas. Quizás su nombre (strings) no sea el más adecuado, pues sólo permite almacenar trozos de texto, sino que igual que una clave se trata de un array de bytes, pudiendo almacenar enteros o datos binarios serializados. La característica principal es que una clave que referencia a una cadena sólo tendrá un elemento de tipo cadena.
- Hashes. La estructura de hash es el equivalente a un diccionario en Python. En lugar de manipular claves individuales directamente, se manipulan los campos de una clave. De nuevo, un campo de una clave podrá ser todo aquello que se pueda representar en un array de bytes.
- Listas. Las listas permiten asociar un array de valores con una única clave. Se tratan de arrays dinámicos a los que se le pueden aplicar las operaciones clásicas sobre estas estructuras de datos (push, pop, trim…).
- Conjuntos. Los conjuntos son exactamente iguales a la listas, con la excepción de que no admiten valores duplicados, permitiendo realizar operaciones como unión o intersección.
- Conjuntos ordenados. Los conjuntos ordenados se parecen a los conjuntos en el sentido de que sólo admiten valores únicos, pero a su vez cada valor es etiquetado con una puntuación. Esta puntuación será utilizada para insertar los elementos en el conjunto de forma ordenada. Por ejemplo, en el simulador se mantiene un conjunto de hashes de bloques generados ordenados por tiempo.

5.2 Estructura

En este apartado se procederá a analizar la estructura y los aspectos más importantes del código del simulador. Se comenzará por analizar la estructura de ficheros del módulo del simulador y se continuará entrando en los detalles de cada uno de los ficheros y clases que lo componen prestando especial atención a los componentes principales que se introdujeron en el capítulo anterior.

Estructura de ficheros

La aplicación que compone el simulador está integrada por una serie de componentes interconectados entre sí: aplicación web, servidor web y simulador. El módulo simulator

(dentro de btcsimulator/server) es el que se estudiará en este apartado y es el que contiene todos los elementos que dan vida al simulador.

Es importante conocer cómo se ha organizado el código para a continuación poder hacer un análisis de forma lógica del mismo. El simulador se ha dividido en 6 ficheros con funcionalidad bien diferenciada como se puede apreciar en la figura 4.

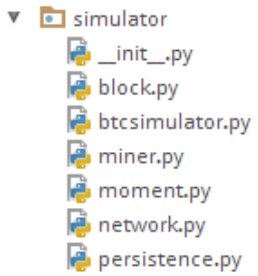

Fig. 4 Estructura de ficheros del simulador

- El fichero block.py contiene la definición de la clase que implementa el modelo de un bloque del protocolo Bitcoin así como los métodos necesarios para su correcto funcionamiento, como es el cálculo de su hash.
- El fichero btcsimulator.py contiene la definición de los diferentes tipos de simulaciones que se pueden crear. En él la clase Simulator se encargará de crear el bloque génesis y de inicializar todos los nodos determinando su capacidad de cómputo y sus conexiones con los demás nodos de forma aleatoria. En este fichero se crea y lanza el entorno de ejecución de simulación. Se han definido tres tipos de simulaciones a partir de los objetivos que se habían marcado para el desarrollo:

  1. Simulación estándar. En este tipo de simulación se crean una serie de nodos honestos que ejecutarán el protocolo Bitcoin durante una serie de días pasados por parámetro.
  2. Ataque del 51%. En esta simulación uno de los nodos tendrá una capacidad de cómputo igual al 51% de la red y su comportamiento no será honesto, sino que rechazará todos los bloques que no hayan sido descubiertos por él.
  3. Selfish attack. En esta simulación uno de los nodos, que tendrá una capacidad de cómputo del 3'% tendrá el comportamiento propio de este ataque, manteniendo en secreto una cadena de forma paralela a la cadena global con el objetivo de incrementar sus beneficios.

- El fichero miner.py contiene el grueso del simulador. Es en él donde se define el comportamiento de los nodos, lo que en definitiva es el protocolo Bitcoin. Aquí se define el comportamiento de los nodos honestos y el de aquellos nodos que pretendan llevar a cabo algún tipo de ataque.
- El fichero momento.py contiene una serie de funciones relacionadas con la gestión del tiempo y la conversión entre diferentes formatos.
- El fichero network.py contiene todas las clases y funciones relacionadas con las comunicaciones de los nodos a través de la red. Esto es los enlaces, los eventos en los que se encapsula la información que se envía por la red y los sockets, que permiten agregar en un solo receptor todos los eventos que se puedan recibir a través de los diferentes enlaces que tiene un nodo.
- El fichero persistence.py se encarga de realizar la conexión con Redis y ofrece una serie de métodos para realizar una serie de operaciones sobre la base de datos como la obtención de un id único para un elemento o el vaciado de la base de datos antes de

comenzar una simulación para garantizar que no haya interferencias con datos de simulaciones anteriores.

Bloques

La clase bloque contenida en el fichero block.py es quizás el componente más sencillo pero de los más necesarios de todo el simulador. Como ha explicado a lo largo de esta memoria, son muchos de los aspectos a nivel de bloque y por debajo (transacciones) que se descartaron de la implementación. Esto ha provocado que un bloque se convierta en un simple contenedor de datos estático con los siguientes atributos que se mapean directamente con el concepto de bloque que se había planteado inicialmente:

- Prev. Hash del bloque predecesor en la cadena.
- Height. Indicador de la dificultad con que fue generada un bloque y que permite determinar qué cadena es la más larga.
- Time. Sello de tiempo de la fecha (dentro de la simulación) en que fue generado el bloque.
- Size. Número aleatorio que pretende representar el tamaño del bloque como reflejo de las transacciones que tiene en su interior.
- Valid. Flag que indica la validez de un nodo. Por el momento todos los bloques son considerados válidos.

Todo modelo del protocolo Bitcoin tendrá un método que permitirá almacenarlo en Redis. Como estructura de datos se utilizará un hash y como clave un identificador único que es generado por la base de datos, con la excepción de los bloques, que su identificador único será su hash. En la 5 figura se puede observar la estructura de un bloque.

```
class Block:
    def __init__(self, prev, height, time, miner_id, size, valid):
        self.prev = prev
        self.height = height
        self.time = time
        self.miner_id = miner_id
        self.size = size
        self.valid = valid
        # When a block is created it is stored in redis
        self.store()

    def store(self):
        key = 'blocks:' + str(sha256(self))
        # Store block in block list
        r.zadd("blocks", self.height, sha256(self))
        # Store the block info
        r.hmset(key, {'prev': self.prev, 'height':self.height, 'time': self.time, 'size': self.size,
        # Store reference block in the miner's blocks set
        r.zadd("miners:" + str(self.miner_id) + ":blocks-mined", self.height, sha256(self))
```

Fig. 5 Estructura de un bloque en el simulador

Enlaces, eventos y sockets

El fichero network.py contiene todos aquellos elementos que son necesarios para realizar las comunicaciones de red entre los nodos. Está compuesto por tres componentes principales: los enlaces, los eventos y los sockets.

La clase Link permite contener la información de la conexión que une a dos nodos. Un enlace tendrá un origen, un destino y un retardo, que permite simular tanto la latencia como el ancho de banda de las conexiones. Por cada par de nodos existirán dos enlaces. Esto es así, ya que en una red de comunicaciones, normalmente el ancho de banda de una comunicación nos es el mismo para las subidas que para las descargas. La estructura de un enlace sería la de la figura 6.

```python
class Link:
    def __init__(self, origin, destination, delay):
        self.origin = origin
        self.destination = destination
        self.id = self.get_id()
        self.delay = delay
        # Store link in database
        self.store()

    def get_id(self):
        return get_id("links")

    def store(self):
        key = "links:" + str(self.id)
        r.sadd("links", self.id)
        r.hmset(key, {"destination": self.destination, "delay": self.delay, 'origin': self.origin})
```

Fig. 6 Estructura de un enlace en el simulador

Otro de los componentes fundamentales para el funcionamiento de la red es la manera en que se envían los datos a través de la misma. En el capítulo anterior se habían definido una serie de eventos que serían los que los nodos se intercambiarían entre sí. Cuando en la red se envía información entre los diferentes nodos, este se debe de encapsular de tal manera que sea fácil poder recuperarla y poder distinguir qué tipo de evento es el que se comunicó a través de la red. Esta función la cumple la clase Event, que tendrá un origen, un destino, un sello de tiempo que se establece cuando se genera, una carga y una acción, que representa uno de los posibles eventos de red como se aprecia en la figura 7.

```python
class Event:
    def __init__(self, destination, origin, time, action, payload):
        self.destination = destination
        self.origin = origin
        self.action = action
        self.payload = payload
        self.time = time
        self.id = self.get_id()
        self.store()

    def get_id(self):
        return get_id("events")

    def store(self):
        key = "events:" + repr(self.id)
        # Store event
        data = {"destination": self.destination, "origin": self.origin, "action": self.action,
        if isinstance(self.payload, Block):
            data['payload'] = sha256(self.payload)
        r.hmset(key, data)
        day = moment.days_passed(self.time)
        r.zadd("events", self.time, self.id)
        r.zadd("days:" + repr(day) + ":events:" + repr(self.action), self.time, self.id)
        r.zadd("days:" + repr(day) + ":events", self.time, self.id)
        r.zadd("miners:" + repr(self.origin) + ":events", self.time, self.id)
```

Fig. 7 Estructura de un evento en el simulador

Por último, dentro de este módulo se encuentra la clase Socket, que se trata permitirá agregar todos los enlaces un una única interfaz que encapsula todos los eventos que se reciben de la red, de tal manera que un nodo sólo tendrá que prestar atención a su Socket, permitiendo que se pueda olvidar del número de enlaces de los que dispone. Por tanto, un socket tendrá conocimiento de todas las conexiones de un nodo y ofrecerá métodos para enviar datos a un único nodo o para realizar una difusión a través de la red. Al mismo tiempo permitirá a un nodo ponerse a la escucha para recibir los eventos de la red. Como esta clase no es propia del protocolo Bitcoin, sino que es una manera de implementarlo, ésta no se está almacenando en la base de datos, pues su información no aportaría valor alguno. En la figura 8 se presenta un detalle de esta clase en el simulador.

```python
class Socket:
    def __init__(self, env, store, miner_id):
        self.miner_id = miner_id
        self.store = store
        self.env = env
        self.links = dict()

    def add_link(self, link):
        self.links[link.destination] = link

    def send(self, value, delay):
        self.env.process(self.process_send(value, delay))

    def process_send(self, value, delay):
        yield self.env.timeout(delay)
        self.store.put(value)

    # Send certain event to a specific miner
    def send_event(self, to, action, payload):
        event = Event(to, self.miner_id, self.env.now, action, payload)
        self.send(event, self.links[to].delay)

    # Broadcast an event to all links
    def broadcast(self, action, payload):
        for to in self.links:
            event = Event(to, self.miner_id, self.env.now, action, payload)
            self.send(event, self.links[to].delay)

    def receive(self, miner_id):
        return self.store.get(filter=lambda event: event.destination == miner_id)
```

Fig. 8 Detalle de la clase Socket

Nodos

El último de los componentes que queda por analizar es el más complejo e importante, ya que son los nodos y su comportamiento los que marcan la ejecución del protocolo y el destino al que se dirige la red. En el fichero miner.py están definidos todos estos comportamientos.

El comportamiento de un nodo honesto se define en la clase Miner. Siguiendo el modelo que se definió durante la fase de diseño, un nodo tendrá un porcentaje de la capacidad de cómputo de la red que le permitirá generar más o menos bloques a lo largo del tiempo, pero manteniendo un ritmo global constante. Además, un nodo tendrá un ratio de verificación que determinará el tiempo que le llevará procesar un bloque en función de su tamaño. Para facilitar el desarrollo, un nodo tendrá un atributo identificando el nodo génesis y otro con el hash de la cabeza de la cadena principal. En lo que se refiere a estructuras del protocolo Bitcoin en sí, un nodo contará con los tres pools de bloques que se definieron en la fase de diseño. Por último, un nodo contará con tres procesos de Simpy que regirán parte de su comportamiento interno y que permitirán detectar cuándo se recibe un nuevo bloque, cuándo se interrumpe el proceso de minado al recibir un evento de red y cuándo reanudar este proceso. En la figura 9 se presenta el proceso de inicialización de un nodo.

```python
def __init__(self, env, store, hashrate, verifyrate, seed_block):
    # Simulation environment
    self.env = env
    # Get miner id from redis
    self.id = self.get_id()
    # Socket
    self.socket = Socket(env, store, self.id)
    # Miner computing percentage of total network
    self.hashrate = hashrate
    # Miner block erification rate
    self.verifyrate = verifyrate
    # Store seed block
    self.seed_block = seed_block
    # Pointer to the block chain head
    self.chain_head = '*'
    # Hash with all the blocks the miner knows about
    self.blocks = dict()
    # Array with blocks needed to be processed
    self.blocks_new = []
    # Create event to notify when a block is mined
    self.block_mined = env.event()
    # Create event to notify when a new block arrives
    self.block_received = env.event()
    # Create event to notify when the mining process can continue
    self.continue_mining = env.event()
    self.mining = None
    # Store the miner in the database
    self.store()
    self.total_blocks = 0
```

Fig. 9 Proceso de inicialización de un nodo

Cuando un nodo es inicializado, este se pondrá a la espera de descubrir nuevos bloques (desde la red o generados por él). Para poder recibir nuevos bloques, el nodo deberá comenzar a minarlos y al mismo tiempo deberá escuchar los eventos que pudieran llegar desde la red. Este proceso se ilustra en la figura 10.

```python
def start(self):
    # Add the seed_block
    self.add_block(self.seed_block)
    # Start the process of adding blocks
    self.env.process(self.wait_for_new_block())
    # Receive network events
    self.env.process(self.receive_events())
    # Start mining and store the process so it can be interrupted
    self.mining = self.env.process(self.mine_block())
```

Fig. 10 Proceso a los que espera un nodo

El proceso de minado se ejecutará de forma ininterrumpida. Es el estado natural en el que se encuentra el nodo en todo momento. Este proceso sólo se parara en el caso de que se descubra o se reciba un nuevo bloque que deba ser procesado. Cuando un nodo descubra un nuevo bloque, éste lo notificará al proceso que se encuentra a la espera de recibir nuevos bloques.

El proceso de minado se puede observar en la figura 11.

```python
def mine_block(self):
    # Indefinitely mine new blocks
    while True:
        try:
            # Determine block size
            block_size = 1024*200*numpy.random.random()
            # Determine the time the block will be mined depending on the miner hashrate
            time = numpy.random.exponential(1/self.hashrate, 1)[0]
            # Wait for the block to be mined
            yield self.env.timeout(time)
            # Once the block is mined it needs to be added. An event is triggered
            block = Block(self.chain_head, self.blocks[self.chain_head].height + 1, self.env.now, self.
            self.notify_new_block(block)
        except simpy.Interrupt as i:
            # When the mining process is interrupted it cannot continue until it is told to continue
            yield self.continue_mining
```

Fig. 11 Proceso de minado en un nodo

Cuando un nodo reciba un nuevo bloque, entonces el proceso de minado será interrumpido, se añadirá el bloque al pool de bloques nuevos y se procesarán todos los bloques como se puede apreciar en la figura 12.

```python
def wait_for_new_block(self):
    while True:
        # Wait for a block to be mined or received
        blocks = yield self.block_mined | self.block_received
        # Interrupt the mining process so the block can be added
        self.stop_mining()
        #print("%d \tI stop mining" % self.id)
        for event, block in blocks.items():
            #print("Miner %d - mined block at %7.4f" %(self.id, self.env.now))
            # Add the new block to the pending ones
            self.blocks_new.append(block)
            # Process new blocks
        yield self.env.process(self.process_new_blocks())
        # Keep mining
        self.keep_mining()
```

Fig. 12 Proceso de espera de nuevos bloques

El procesado de los nuevos bloques implica iterar sobre el pool de los bloques nuevos, verificar la validez de estos bloques (se comprueba si se tiene la información del bloque anterior o si la dificultad del bloque es la correcta comparándolo con el anterior. Procesar un bloque lleva un tiempo a un nodo. Puede ocurrir que no se tenga información suficiente de un bloque para procesarlo en el momento, por lo que se deberá pedir el bloque a la red mediante un BLOCK_REQUEST. Este proceso se ilustra en la figura 13.

```python
def process_new_blocks(self):
    blocks_later = []
    # Validate every new block
    for block in self.blocks_new:
        # Block validation takes some time
        yield self.env.timeout(block.size / self.verifyrate)
        valid = self.verify_block(block)
        if valid == 1:
            self.add_block(block)
        elif valid == 0:
            #Logger.log(self.env.now, self.id, "NEED_DATA", sha256(block))
            self.request_block(block.prev)
            blocks_later.append(block)
    self.blocks_new = blocks_later
```

Fig. 13 Procesado de bloques en un nodo

Cuando se añade un bloque a la cadena, se comprueba si éste da lugar a una cadena más larga, que pasará a convertirse en la cadena principal como se ilustra en la figura 14.

```python
def add_block(self, block):
    # Add the seed block to the known blocks
    self.blocks[sha256(block)] = block
    # Store the block in redis
    r.zadd("miners:" + str(self.id) + ":blocks", block.height, sha256(block))
    # Announce block if chain_head isn't empty
    if self.chain_head == "*":
        self.chain_head = sha256(block)
    # If block height is greater than chain head, update chain head and announce new head
    if (block.height > self.blocks[self.chain_head].height):
        self.chain_head = sha256(block)
        self.announce_block(block)
```

Fig. 14 Proceso de adición de un bloque a la cadena principal

El proceso que se encuentra a la escucha de los eventos de red esperará a que el socket reciba algún tipo de información y, en función del tipo de evento recibido y siguiendo el diseño del protocolo, realizará la acción adecuada. El conjunto de acciones a realizar se observa en la figura 15.

```python
def receive_events(self):
    while True:
        # Wait for a network event
        if len(self.socket.links) == 0:
            return
        data = yield self.socket.receive(self.id)
        if data.action == Miner.BLOCK_REQUEST:
            # Send block if we have it
            if data.payload in self.blocks:
                self.send_block(data.payload, data.origin)
        elif data.action == Miner.BLOCK_RESPONSE:
            self.notify_received_block(data.payload)
        elif data.action == Miner.HEAD_NEW:
            # If we don't have the new head, we need to request it
            if data.payload not in self.blocks:
                self.request_block(data.payload)
```

Fig. 15 Proceso de escucha de eventos de red

Un nodo también tiene capacidad para enviar eventos. Para ello se definen funciones que facilitan la creación de los eventos en función de su tipo y también se crean funciones que comunican con el socket para enviar o difundir un determinado evento. Estas funciones se ilustran en la figura 16.

```python
# Announce new head when block is added to the chain
def announce_block(self, block):
    self.broadcast(Miner.HEAD_NEW, sha256(block))

# Request a block to all links
def request_block(self, block, to=None):
    #Logger.log(self.env.now, self.id, "REQUEST", block)
    if to is None:
        self.broadcast(Miner.BLOCK_REQUEST, block)
    else:
        self.send_event(to, Miner.BLOCK_REQUEST, block)

# Send a block to a specific miner
def send_block(self, block_hash, to):
    # Find the block
    block = self.blocks[block_hash]
    # Send the event
    self.send_event(to, Miner.BLOCK_RESPONSE, block)

# Send certain event to a specific miner
def send_event(self, to, action, payload):
    self.socket.send_event(to, action, payload)

# Broadcast an event to all links
def broadcast(self, action, payload):
    self.socket.broadcast(action, payload)
```

Fig. 16 Funciones de comunicación de un nodo

Como todo elemento del simulador, los nodos también tendrán un identificador único y se almacenarán en Redis para realizar un posterior análisis de su información.

Por último, se definen dos subclases que heredan de Miner y que pretenden simular el comportamiento de nodos que ejecuten un selfish attack o que tengan más de un 50% de la capacidad de cómputo. Se sobrescribe en ellos la función add_block, que es en la que se determina la manera en la que se añaden los bloques a la cadena principal. El proceso se observa en la figura 17.

```
class BadMiner(Miner):

    # A bad miner (with more than 50% of network computing power
    # will ignore all blocks no mine by self
    def add_block(self, block):
        # Add the block to the known blocks
        self.blocks[sha256(block)] = block
        # Store the block in redis
        r.zadd("miners:" + str(self.id) + ":blocks", block.height, sha256(block))
        # Announce block if chain_head isn't empty
        if self.chain_head == "*":
            self.chain_head = sha256(block)
        # Ignore all blocks that are not mined by the bad miner
        if block.miner_id != self.id:
            return
        # If block height is greater than chain head, update chain head and announce new head
        if block.height > self.blocks[self.chain_head].height:
            self.chain_head = sha256(block)
            self.announce_block(block)
```

Fig. 17 Implementación de un nodo que ejecuta un ataque del 51%

Simulación

Por último, la clase Simulator se encarga de la ejecución de la simulación. Para ello tendrá que crear los nodos asignándoles una capacidad de cómputo de forma aleatoria y a su vez los interconectará también de forma aleatoria. Por último se inicializan los nodos y se da comienzo a la simulación dentro del entorno controlado de Simpy. Antes de comenzar una simulación se vacía la base de datos para evitar conflictos con simulaciones anteriores y al término de la misma se publica un mensaje en Redis para notificar la finalización de la misma y se almacenan algunos datos del estado final de la misma. El proceso de creación de nodos se puede apreciar en la figura 18

```
miners = []
# This dict is used to store the connections between miners, so they are not created twice
connections = dict()
for i in range(0, miners_number):
    miner = Miner(env, store, hashrates[0,i] * Miner.BLOCK_RATE, Miner.VERIFY_RATE, seed_block)
    miners.append(miner)
    connections[miner] = dict()
# Randomly connect miners
for i, miner in enumerate(miners):
    miner_connections = numpy.random.choice([True, False], miners_number)
    for j, miner_connection in enumerate(miner_connections):
        # Onlye create connection if miner is not self and connection does not already exist
        if i != j and miner_connection == True and j not in connections[miner] and i not in connections[miners[j]]:
            # Store connection so its not created twice
            connections[miner][j] = True
            connections[miners[j]][i] = True
            Miner.connect(miner, miners[j])
```

Fig. 18 Proceso de creación de los nodos y sus enlaces

Resolución de problemas

Durante el proceso de desarrollo del simulador se produjeron una serie de retos, problemas e inconvenientes que hubo que superar y que en cierta medida han condicionado la estructura final que tiene el simulador.

El principal y más importante problema que se tuvo que afrontar giraba en torno a Simpy. Esta biblioteca ha sido vital para el desarrollo del simulador. Sin embargo, hay algunos hechos que durante un tiempo pusieron en duda si se llegaría a dar con un producto funcional tal y como se deseaba.

Las pruebas que se iban haciendo a medida que se iba desarrollando el simulador, por cuestiones de eficiencia y comprensión, se realizaban con 2 o 3 nodos como mucho. Esto fue un grave error, pues cuando ya se tenía un primer simulador implementado resultó que muchos de los nodos dejaban de recibir eventos de determinados enlaces. Este comportamiento parecía totalmente aleatorio, pero tras un complejo proceso de depuración se concluyó que la forma en que se escuchaban los eventos de cada uno de los eventos provenientes de los enlaces se quedaba colgada. Las funciones de Simpy que se usaban para ello hacían que muchos eventos fueran pasados por alto.

Por tanto, había que cambiar la implementación del sistema de escucha de eventos. Fue entonces cuando apareció la idea del socket. Simpy permite utilizar un recurso compartido para enviar información entre diferentes nodos y aplicar filtros a los eventos que se disparan en el recurso para que sólo se disparen en un proceso a la escucha si se cumplen determinadas condiciones. De esta manera cada nodo tendría un socket y todos los sockets de la red compartirían un recurso a través del que se podría enviar información. Usando los filtros al recurso y los enlaces de cada nodo se conseguiría capturar todos los eventos que se enviraran por la red y sólo serían capturados por aquellos nodos que los debieran capturar.

La idea parecía que podría funcionar y entonces se decidió implementarla. Una vez hecha la implementación se pasó a realizar las primeras pruebas, que no resultaron satisfactorias. La implementación parecía correcta. El fallo era de extrañar. Buscando en el issue tracker del repositorio de Simpy [48] se aprendió que se trataba de un bug y que no se sabía cuándo sería corregido.

Las ideas se acababan y el tiempo del que se disponía cada vez era menor. Por suerte, el día 14 de mayo se publicó una actualización de la biblioteca que corregía el bug que afectaba al simulador. Tras la actualización de la biblioteca, el programa se ejecutaba de forma correcta, pero por un momento se temió por que no se llegara a tener un simulador totalmente funcional.

5.3 Estructura de datos en Redis

Una vez termina la simulación, se contará con un conjunto de datos en Redis. Se ha intentado crear una estructura lógica y coherente de los datos que facilite tanto su acceso como su comprensión. A continuación se presenta la manera en que se ha organizado la información en Redis. Si se observa el código del simulador se apreciará que además de almacenar todos aquellos componentes básicos (nodos, enlaces y bloques) también se almacenan en diferentes conjuntos las relaciones que se establecen entre estas entidades (por ejemplo los bloques que han sido generados por un nodo en concreto). Además, también se almacenan de forma ordenada todos los eventos que se intercambian en la red, por lo que a partir de estos datos, sería muy sencillo replicar una simulación. Es necesario aclarar que en los próximos párrafos cuando se haga uso de la palabra hash se estará refiriendo a la estructura de datos de Redis (diccionario) y no a las funciones hash. A continuación se presenta el listado de las diferentes claves y su contenido. Un trozo de una clave rodea por “<” y “>” indicará que es un parámetro variable.

- ids: - Todas las claves que empiecen por esta cadena hacen referencia a un valor que se irá incrementando a medida que se accede a él y permitirá obtener un identificador único para cada elemento de la simulación.
    - ids:miners. Devuelve un identificador único para un nodo.
    - ids:links. Devuelve un identificador único para un enlace.
    - ids:events. Devuelve un identificador único para un evento de red.
    - Los bloques no necesitan un id, ya que su hash es único.
- Conjuntos de todas las entidades. Para poder obtener un listado de todos los elementos de una determinada clase se han definido una serie de conjuntos que

contienen los identificadores de todos los elementos que existen de una determinada clase:
    - Miners. Conjunto con los ids de todos los nodos.
    - Links. Conjunto con los ids de todos los enlaces.
    - Events. Conjunto con los ids de todos los eventos.
    - Blocks. Conjunto con los ids de todos los bloques descubiertos.
    - Days. Conjunto con los días que transcurrieron en la simulación.
- miners:<id> - Toda clave que empiece por esta cadena hará referencia a la información del nodo con id = <id>.
    - miners:<id>:blocks-mined. Conjunto de hashes de bloques ordenados por altura que han sido descubiertos por el nodo seleccionado.
    - miners:<id>:events. Conjunto de ids de eventos ordenados por el tiempo de generación de los eventos generados por el nodo seleccionado.
    - miners:<id>. Hash con toda la información de un nodo.
    - miners:<id>:links. Conjunto de los enlaces que tiene un nodo.
    - miners:<id>. Conjunto de los bloques que un nodo tiene ordenados por su altura.
    - Miners:<id>:head. Hash del último bloque de la cadena del nodo seleccionado al terminar la simulación.
- Links:<id> - Toda clave que empiece por esta cadena hace referencia a la información del enlace con id = <id>.
    - Links:<id>. Hash con toda la información de un enlace.
- Events:<id> - Toda clave que empiece por esta cadena hace referencia a la información del evento con id = <id>.
    - Events:<id>. Hash con toda la información de un evento
    - Events:types. Conjunto con todos los tipos de eventos existentes.
- Days:<dia>:<key>. Las claves de este tipo serán conjuntos que devuelvan aquellos elementos que ocurrieron en un determinado día.
    - Days:<dia>:events. Conjunto con los ids de eventos que ocurrieron en un día determinado.
    - Days:<dia>:events:<type>. Conjunto con los ids de eventos del tipo = <type> que ocurrieron un determinado día.

Con este sistema de nomenclatura de las claves resulta bastante sencillo recuperar toda la información que se desee de la simulación. El hecho de que toda la información esté en la RAM hace que los tiempos de respuesta sean extremadamente cortos.

5.4 Rendimiento del simulador

En este apartado se pretende aportar una serie de detalles técnicos sobre la ejecución del simulador y el tiempo que se tarda en ejecutar en función del número de nodos y de días.

Aunque Simpy es compatible con las versiones más recientes de Python, el desarrollo se hizo en la versión 2.7 ya que una de las bibliotecas utilizadas en el servidor web para poder escuchar los mensajes que se pasen a través de Redis solo está soportada hasta esta versión del lenguaje.

Para la gestión de las diferentes dependencias del simulador y de los componentes de la aplicación escritos en Python se utiliza pip [49]. En la raíz del código hay un fichero requirementes.txt con todas las dependencias y sus versiones.

En la raíz del código también se puede encontrar un fichero readme en el que se dan instrucciones más detalladas de los pasos necesarios para poder ejecutar el simulador. Por tanto, se obviará dar esa explicación en este documento, sino que sólo se nombrarán los aspectos más importantes.

El simulador se puede ejecutar como un script de Python cualquiera. El fichero btcsimulator.py del módulo simulator lanzará por defecto una simulación de 3 nodos y 1 día. En la figura 19 se puede apreciar los ficheros principales de la aplicación.

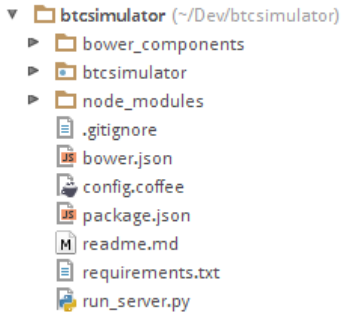

Fig. 19 Ficheros principales del simulador

Las pruebas realizadas con el simulador se llevaron a cabo en un equipo con un procesador Intel Core i7 a 3.50GHz y con 16GB de RAM. El Sistema Operativo utilizado era Elementary Os. Este equipo tiene unas características más que aceptables y como se podrá ver en la figura 20, a pesar de esto hecho, el uso de CPU y durante la ejecución de una simulación es bastante relevante.

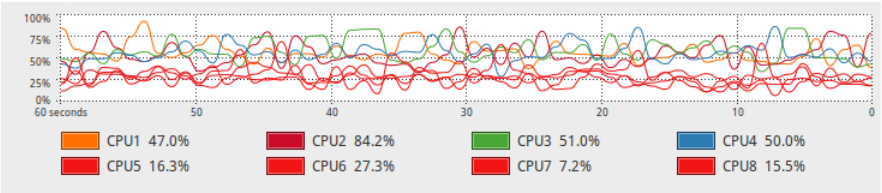

Fig. 20 Consumo de CPU durante la ejecución de una simulación

La capacidad de la memoria RAM influye en el tipo de simulación que se puede llegar a ejecutar, al estar limitado el tamaño de la base de datos de Redis al espacio disponible en memoria RAM.

El tiempo que se tarda en ejecutar la simulación depende de las dimensiones de la misma, es decir del número de días que se ejecute y del número de nodos. Pero también depende del número de conexiones que se establecen entre ellos. De esta manera, una misma simulación de 3 días y 3 nodos puede tardar desde un par de segundos hasta más de 10.

A medida que se incrementa el número de estos parámetros el tiempo va creciendo de forma exponencial. La tabla 1 presenta una relación entre el tiempo de ejecución, el número de nodos, los días de ejecución y número de claves diferentes almacenados en Redis.

Número de nodos	Días de simulación	Tiempo de ejecución	Claves en Redis
3	10	15s	13614
3	20	18s	28506
10	3	42s	101406
17	3	608s	446517
10	10	132s	322491
20	20	11660s	5125779

Tabla 1 Tiempos de ejecución del simulador

Si se compara estos tiempos de ejecución con los del simulador en Python inicial, se podrá observar que son de la misma magnitud, llegando a mejorarlos en muchas ocasiones, por lo

que se consiguió desarrollar un simulador mejor estructurado, más escalable, con algún aspecto extra del protocolo Bitcoin sin perder en eficiencia y eficacia.

6 Desarrollo del servidor y la aplicación web

El desarrollo de este trabajo desde un inicio se había divido en dos módulos principales. Por un lado, está el simulador y, por el otro una interfaz que permitiera analizar los resultados de una simulación para poder sacar conclusiones de todo lo ocurrido. En este capítulo se procederá a analizar la estructura y funcionamiento de la API y la aplicación web desarrolladas con el objetivo de satisfacer este requisito del simulador.

Partiendo de la información que se tiene almacenada en Redis, se pretendía desarrollar una API REST [50] que sirviera toda la información a una aplicación web que se encargaría de mostrar los resultados de la ejecución de la simulación al usuario.

Esta aplicación web presentará una interfaz amigable desde la que se procesarán y tratarán los datos devueltos por el servidor para mostrar un conjunto de gráficas, tablas y otras visualizaciones que de un simple vistazo permitirán hacerse una idea de cómo ha ido la simulación.

Como último objetivo, se había marcado la posibilidad de poder lanzar las simulaciones desde la propia aplicación web. Esta funcionalidad daría un gran aporte de usabilidad al simulador. Para poder implementar esta funcionalidad es necesario disponer de un sistema que permita lanzar procesos en segundo plano, ya que las simulaciones pueden tardar bastante tiempo en ejecutarse en función de la configuración seleccionada y, no se puede permitir que el servidor web se bloquee y no pueda desempeñar su función, que es la de servir la información. Se verá que el uso de Redis ha posibilitado la implementación de esta funcionalidad.

Cuando se lance una simulación desde la aplicación web, esta quedará a la espera de que termine la simulación. De alguna manera, esta debe conocer en qué momento termina la ejecución de este proceso para poder permitir al usuario analizar los datos. Usar el protocolo HTTP para realizar esto no es recomendable, pues se trata de un protocolo sin estado, en el que no se permite realizar una comunicación directa del servidor a un cliente sin ser este último el que comienza la aplicación. Esto implicaría estar constantemente realizando consultas al servidor para determinar si la tarea ya terminó o no. Es algo totalmente ineficiente y existen mejores aproximaciones a este problema. El uso de Websockets [43] permite establecer un canal de comunicaciones en ambos sentidos entre aplicación y servidor, posibilitando el envío de información (PUSH) desde el servidor al cliente. El uso de estos protocolos de comunicación permite que no se sature tanto el servidor web. Además, este diseño favorece la escalabilidad que se había marcado como objetivo. Los tres componentes de la aplicación (simulador, servidor y aplicación web) se pueden ejecutar en máquinas diferentes y aun así el simulador seguirá funcionando correctamente.

En el resto de este capítulo, de forma análoga al capítulo anterior, se introducirá el conjunto de tecnologías utilizadas para desarrollar la aplicación, se hablará acerca de la información que el servidor es capaz de devolver y se introducirá tanto la estructura del servidor como de la aplicación web entrando en algunos detalles de la implementación de las visualizaciones.

6.1 Tecnologías utilizadas

Desde el comienzo del desarrollo del simulador, siempre se tuvo la intención de implementar todos aquellos módulos como fuera posible en Python con el objetivo de tener una uniformidad en el desarrollo. Sin embargo, en el momento en el que se introduce el desarrollo de una aplicación web, se está hablando indirectamente de desarrollar utilizando HTML, CSS y especialmente JavaScript si se pretende dotar a la aplicación de un gran dinamismo que la convierta más rápida y usable. Es por esto que a pesar de que el servidor web está escrito en Python, todo el módulo de visualización y la interfaz están desarrolladas en estos lenguajes tan comunes en el entorno web.

Flask

Todo el servidor web, que se encarga tanto de servir los ficheros estáticos de la aplicación web, como la información proveniente de Redis a través de una API REST, se encuentra escrito en Python. Este lenguaje cuenta con una gran variedad de frameworks para el desarrollo de aplicaciones web como son Django, Pyramid o Flask. Para desarrollar el servidor se eligió Flask [51], que se trata de un microframework para el desarrollo de aplicaciones web basado en un sistema de extensiones. Con una API muy sencilla que permite desarrollar aplicaciones de forma extremadamente rápida, este framework incluye un gran soporte para el desarrollo de aplicaciones RESTful, que es la manera en la que se había decidido que se implementaría la API del servidor web por su facilidad e integración con las aplicaciones web en comparación con otros protocolos del mismo nivel como SOAP.

Este framework incluye todos los componentes que se necesitan para desarrollar la aplicación deseada. Además, presenta grandes similitudes con Sinatra [52], un framework de aplicaciones web en Ruby con el que ya se tenía una experiencia previa. En próximos apartados se verá que se desarrolló un servidor con bastante funcionalidad en muy pocas líneas de código.

Stack de aplicaciones HTML 5

Para el desarrollo de la aplicación web, que se quería que fuera dinámica, rápida, usable y que permitiera mostrar diferentes visualizaciones de los datos, no se podían utilizar los métodos clásicos de desarrollo de aplicaciones web compuestas por diferentes páginas HTML estáticas. Era necesario aportar dinamismo y frescura a la interfaz y eso se traduce al desarrollo de una aplicación web HTML 5.

En los últimos años, la evolución y popularidad de estas tecnologías no ha dejado de crecer hasta el punto que JavaScript se ha convertido en el lenguaje en el que más se programa a día de hoy. Esto ha hecho que se haya creado un gran ecosistema de frameworks, bibliotecas y utilidades que permiten crear aplicaciones realmente poderosas en muy breves periodos de tiempo. Esta diversidad ha hecho que para el desarrollo de este tipo de aplicaciones existan muy variadas alternativas que se pueden combinar de múltiples maneras para el desarrollo de una aplicación web. Es lo que se conoce como el stack (pila) de tecnologías para aplicaciones HTML 5.

El stack utilizado para el desarrollo de la aplicación web del simulador es el siguiente:

- Coffeescript. Coffeescript [53] se trata de un lenguaje de programación que compila a JavaScript. Añade numerosos componentes inspirados en Ruby y Python que facilitan su lectura, comprensión y que reducen mucho la extensión del código final, haciendo que se puedan conseguir desarrollos mucho más rápidos.
- Brunch. Brunch [54] es una herramienta para el desarrollo de aplicaciones HTML 5 que se puede ver como el pegamento entre todos los componentes. Se encarga de compilar los scripts, plantillas y estilos, minificándolos, optimizándolos y uniéndolos en un único fichero para ocupar el menor espacio posible. Además facilita el desarrollo mediante un análisis de código para mostrar fallos y el continuo refresco de la página del navegador cada vez que se produce un cambio en el código.
- Bower. Bower [55] es un gestor de dependencias para aplicaciones web. Es el equivalente a Maven en Java o Npm en Node. Permite mantener un control de todas las bibliotecas de terceros utilizadas en la aplicación y realizar una gestión de todas estas dependencias de forma ordenada, coherente y eficiente.

Además de esta pila de tecnologías sobre la que se construye esta aplicación, también se han utilizado una serie de bibliotecas y frameworks que han acelerado el desarrollo y que han

hecho posible entre otros aspectos la creación de las visualizaciones que permiten observar todo lo acontecido durante la simulación. De todas estas bibliotecas merecen especial atención pos su importancia las 3 siguientes.

- Backbone.js y Chaplin. Backbone.js [56] se trata de un framework de aplicaciones web que permite separar el contenido de las mismas siguiendo el patrón MV*, permitiendo reciclar mucho código y evitar el conocido código Espagueti. Inicialmente se pretendía realizar la webapp utilizando Angularjs [57], otro framework de aplicaciones web que goza de gran popularidad actualmente. Sin embargo, apenas se tenían conocimientos acerca de este lenguaje y, puesto que el tiempo era un factor en contra, se decidió finalmente utilizar Backbone, del que se disponen amplios conocimientos y permitiría avanzar más rápido. Chaplin [58] es un framework escrito sobre Backbone que añade una serie utilidades y complementos como un agregador de eventos.
- D3.js. Una de las razones para el desarrollo de esta aplicación web era la creación de toda una serie de visualizaciones que permitiera observar los resultados. Por esto, estas visualizaciones debían ser de gran calidad, pues son el principal valor añadido que se consigue con la aplicación web. Existen múltiples bibliotecas escritas en JavaScript para la representación de gráficas, pero si una ha destacado entre el resto últimamente, esta es D3.js [59], una biblioteca para la manipulación de documentos basada en los datos (Data-Driven-Documents) que permite dar vida a datos usando estándares web como HTML, SVG y CSS. Con un lenguaje declarativo permite crear visualizaciones interactivas de forma muy sencilla y que resultan muy atractivas.
- Flat-UI. Fuera de toda duda, cualquier aplicación que se preste y disponga de una interfaz, entrará por los ojos independientemente de su funcionalidad. Por este motivo es necesario construir una interfaz de usuario atractiva desde el punto de vista visual, para conseguir mejorar la usabilidad del simulador. Para ello se han hecho uso de dos frameworks CSS muy populares y que han permitido dar lugar a una interfaz bonita y usable. Estos son Bootstrap [60], sobre el que se construye el segundo de ellos: Flat-UI [61]. Se tratan de una serie de componentes de interfaz y herramientas que permiten diseñar una web con un buen aspecto visual en muy poco tiempo. Una de las grandes ventajas es que están por defecto preparados para ser responsive y adaptarse al tamaño de la pantalla del dispositivo con el que se está visualizando la web.

6.2 Métodos de la API

Antes de comenzar a analizar la estructura del simulador, es primordial indicar qué es lo que hace, es decir, qué datos devuelve para que puedan ser consumidos por la aplicación web. Esta es su función primordial y, por eso merece la pena pararse para analizar qué es lo que hace. La información sólo se sirve para que pueda ser leída, es decir, no se permite crear nuevos datos ni actualizarlos, por lo que todos los métodos listados a continuación, son accedidos mediante un HTTP GET. Todos los datos de la API son devueltos en formato JSON.

- /miners. Devuelve un listado con la información de todos los nodos de la red incluyendo sus enlaces.
- /miners/:id. Devuelve la información del nodo con id = :id. En la figura 21 se observa un ejemplo.

```
- {
    blocks: 326,
    blocks_mined: 11,
    hashrate: "0.030943237598440837",
    head: "c406114c2d91a14c5385830226971a7c463979d93902f33e8ea608add3633e18",
    id: "1",
  - links: [
      - {
            delay: "0.02",
            destination: "2",
            origin: "1"
        }
    ],
    verifyrate: "204800"
  },
```


Fig. 21 Información de un nodo devuelta por la API

- /miners/:id/links. Devuelve la información de los enlaces del nodo con id = :id. En la figura 22 se observa un ejemplo.


```
- {
    delay: "0.02",
    destination: "2",
    origin: "1"
  }
```


Fig. 22 Información de un enlace devuelta por la API

- /miners/:id/blocks/page/:page. Devuelve el listado con todos los bloques tiene un nodo. Dado que este listado puede ser muy grande, es necesario paginar los resultados. En la figura 23 se observa un ejemplo.


```
- {
    hash: "c406114c2d91a14c5385830226971a7c463979d93902f33e8ea608add3633e18",
    height: "325",
    miner: "2",
    prev: "040b6e49c0ea58f1c660f5f2659862ad23b4dd5723942b9bc58479661cf90cba",
    size: "116990.98084642841",
    time: "258882.41906828011",
    valid: "1"
  },
```


Fig. 23 Información de un bloque devuelta por la API

- /miners/:id/blocks-mined/page/:page. Devuelve el listado con todos los bloques que han sido descubiertos por un nodo en concreto. También es necesario paginar los resultados.
- /blocks/page/:page. Devuelve un listado con todos los bloques existentes ordenados por altura. Dada su cantidad, es necesario paginar los resultados.
- /blocks/:id. Devuelve toda la información de un bloque a partir de su hash.
- /events/page/:page. Devuelve un listado con todos los eventos ocurridos durante la simulación. Dada su cantidad, es necesario paginarlos.
- /events/:id. Devuelve toda la información de un evento a partir de su id.
- /days. Devuelve el listado con todos los días de la aplicación.

- /days/:id/events/page/:page. Devuelve el listado con todos los eventos ocurridos en un día de la simulación. Dada su cantidad también es necesario paginarlos. En la figura 24 se presenta un ejemplo de evento.


```
- {
    action: "3",
    destination: "1",
    id: "1",
    origin: "2",
    payload: "a800f025689f6c7d2ab3058c62c5841097bc49ce9cd8efb65046619378464458",
    time: "47.551849971721772"
  },
```


Fig. 24 Información de un evento devuelta por la API

- /miners/:id/events/page/:page. Devuelve un listado con todos los eventos que un nodo ha generado en un día. Dada su cantidad, es necesario paginar los resultados.
- /links. Devuelve un listado con todos los enlaces que hay entre los nodos.
- /links/:id. Devuelve la información del enlace con id = :id.
- /chain/:id. Devuelve la cadena de bloques a la que pertenece el bloque con hash = :id. Lo que hace este método es ir recorriendo hacia atrás la cadena hasta llegar al bloque original.
- /summary. Devuelve un resumen de la simulación con el número de nodos, enlaces, bloques días y eventos. En la figura 25 se muestra un ejemplo de un resumen de una simulación.


```
- data: {
    blocks: 474,
    days: 3,
    events: 1300,
    links: 2,
    miners: 3
  }
```


Fig. 25 Resumen de una simulación

Con toda esta información devuelta con el servidor se podrán construir todas las visualizaciones necesarias desde la aplicación web, que constantemente hará peticiones a este servidor para nutrirse de los datos a medida que los vaya necesitando.

6.3 Estructura del servidor

En este apartado se analizará la estructura del servidor web y sus principales componentes. Para ello se comenzará revisando la estructura de ficheros utilizada. Todos los componentes del servidor web se encuentran dentro del módulo btcsimulator/server y a su vez la definición de los manejadores de todas las rutas se realiza desde el módulo controllers. En la figura 26 se puede observar esta estructura.

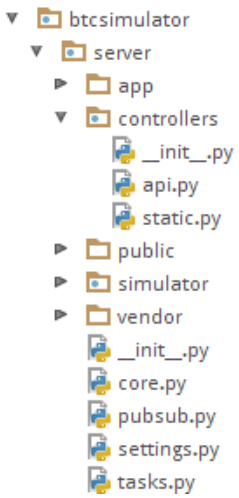

Fig. 26 Estructura de ficheros del servidor web

- En el fichero __init__.py se inicializa la aplicación de Flask y se cargan el resto de componentes de la aplicación. Así desde este fichero se expone la aplicación para que pueda ser utilizada desde otros módulos. En la figura 27 se muestra este proceso de carga.

```
app = Flask(__name__, static_url_path='', static_folder='public')
app.config.from_object(settings)
app.url_map.strict_slashes=False


import core
import tasks
import controllers
```

Fig. 27 Proceso de inicialización del servidor web

- El fichero settings.py contiene todas las variables de configuración utilizadas en la aplicación. Los datos de conexión al servidor de Redis o el espacio de nombres utilizado para los mensajes que se publiquen a través de Redis.
- En el fichero core.py se definen los componentes principales de la aplicación, se crea la conexión a la base de datos, se instancia un logger para tener constancia de los errores y eventos que ocurran durante la ejecución del servidor. Además, se define un decorator para los métodos de la URL que se encarga de la gestión de las políticas Cross Origin, con el objetivo de permitir que los datos de la API puedan ser consultados por aplicaciones de terceros que no sean servidas en el mismo dominio que el servidor. Por último, en este fichero también se definen los procesos que permitirán la ejecución del simulador en segundo plano sin bloquear el servidor web. Este aspecto será estudiado más adelante con algo más de detenimiento.
- En el fichero pubsub.py se definen todos los métodos que permiten que la aplicación web y el servidor se puedan conectar mediante el uso de WebSockets y así poder notificar a la aplicación cuándo ha terminado la ejecución de una simulación. Se estudiará con más detenimiento en apartados venideros.
- En el fichero tasks.py se definen los trozos de código (tareas) que se ejecutarán en segundo plano. Lo que se hace es lanzar una simulación u otra en función de los parámetros seleccionados. En la figura 28 se muestra un ejemplo de creación de tareas de celery.

```
@celery.task(name="tasks.simulation")
def start_simulation_task(miners, days, type):
    if type == 'standard':
        Simulator.standard(miners, days)
    elif type == 'fifty-one':
        Simulator.fifty_one(miners, days)
    elif type == 'selfish':
        Simulator.selfish(miners, days)
```

Fig. 28 Creación de una tarea con celery

- Dentro de la carpeta controllers hay dos ficheros. Por un lado está el fichero static.py, que se encarga de servir todos los ficheros estáticos que componen la aplicación web. Las rutas de la aplicación se manejan desde JavaScript. Es por eso que para todas las rutas se devuelve siempre el mismo fichero, como se puede apreciar en la figura 29.

```
@app.route('/')
@app.route('/simulation')
@app.route('/stats/network')
@app.route('/stats/blocks')
@app.route('/stats/explorer')
def root():
    return app.send_static_file('index.html')
```

Fig. 29 Declaración de rutas estáticas del servidor

El otro fichero es api.py. En él se definen todas las rutas que se introdujeron en el apartado anterior. Para poder devolver datos, será necesario hacer diferentes consultas a Redis y transformar las estructuras que son devueltas por Redis para que sean un poco más amigables y contengan más información. Por ejemplo las listas y conjuntos se sustituyen con toda la información de una entidad en lugar de devolver solo su identificador. Además aquellos listados que se prevea que contendrán una gran cantidad de datos, como son los bloques o los eventos, serán paginados para ahorrar tráfico de datos y no sobrecargar la aplicación web con cantidades de información que no será capaz de manejar. En la figura 30 se puede observar un ejemplo de una de estas rutas del servidor en el que se devuelve la cadena que tiene un determinado bloque como extremo.

```
# We just return first 10 blocks. Since it is a linked list it
# is quite easy to get the next page
@app.route('/chain/<string:head>', methods=['GET'])
@crossdomain(origin='*')
def chain(head):
    data = []
    count = 0
    while head != None and count < 100:
        block = get_block(head)
        if block['hash'] != "None":
            data.append(block)
        prev = r.hget("blocks:" + head, "prev")
        head = prev
        if head is None:
            a = 1
        count += 1
    return jsonify(data=data)
```

Fig. 30 Ruta de la API que devuelve la cadena de un bloque

Con esto ya se tiene una idea global del funcionamiento y funcionalidad del servidor y se puede pasar a examinar la estructura y contenido de la aplicación web.

6.4 Estructura de la aplicación web

Al comienzo de este capítulo se indicaba que esta aplicación web se había desarrollado utilizando la herramienta Brunch para la automatización de muchos de los procesos necesarios a la hora de desarrollar una aplicación web. Basado en el paradigma de Convention Over Configuration [62], permite gestionar completamente un proyecto sin apenas utilizar ficheros de configuración. Es por eso que la estructura de ficheros de la aplicación web viene condicionada por estas convenciones. Todo el contenido de la aplicación web se encuentra en dos carpetas: app y public. En app se encuentra el código fuente de la misma y en public se encuentra el resultado que produce Brunch cuando compila la aplicación. Su contenido, que es parecido al de app no es relevante para el entendimiento de la aplicación. Por eso, este apartado se centrará en comentar los principales elementos contenidos en la carpeta app.

Antes de pasar a la explicación de la estructura de la aplicación, es necesario destacar fichero fundamental. Este es el bower.json. Este fichero es el fichero de configuración de Bower y en él se indican todas las dependencias de bibliotecas externas que la aplicación necesita para funcionar. Brunch se encarga de la instalación de las mismas. Puede ocurrir que algunas de las bibliotecas que se quieran utilizar en el proyecto no estén soportadas por Bower. Estas se sitúan en la carpeta vendor dentro de server. En la figura 31 se presentan las dependencias de Bower de la aplicación

```
"dependencies": {
  "backbone": "1.1.2",
  "chaplin": "1.0.0",
  "console-polyfill": "0.1.0",
  "lodash": "2.4.1",
  "jquery": "2.0.2",
  "normalize-css": "2.1.2",
  "h5bp-helpers": "0.1.0",
  "momentjs": "2.4.0",
  "backbone-pageable": "1.4.1",
  "backbone-relational": "0.8.7",
  "backbone.stickit": "0.6.3",
  "Flat-UI": "2.1.1",
  "jquery.transit": "~0.9.9",
  "i18next": "1.7.1",
  "nprogress": "0.1.2",
  "sugar": "1.4.1",
  "d3": "~3.4.8",
  "nvd3": "~1.1.15-beta"
},
```

Fig. 31 Dependencias de la aplicación web

El contenido de la carpeta app es el de la figura 32.

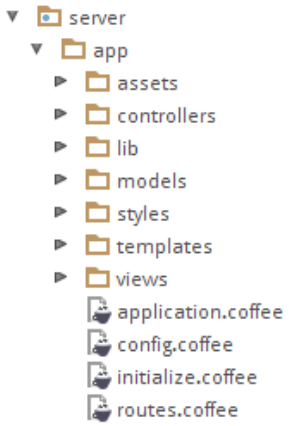

Fig. 32 Contenido de la carpeta app de la aplicación web

Backbone.js hace uso de un patrón Modelo Vista Controlador algo adulterado para dotar de estructura a las aplicaciones. Es por eso que la estructura de ficheros presenta muchas similitudes con muchas otras aplicaciones web.

Una aplicación de Backbone es del tipo Single Page, es decir, sólo existe un único fichero HTML y todo el contenido es generado desde JavaScript, que se encarga de cambiar las URLs del navegador cuando sea necesario sin abandonar en ningún momento la página en la que se está (Pushstate).

El punto de entrada a la aplicación es initialize.coffee, donde se instanciará la aplicación de Backbone que será la que tome el mando a partir de ese momento. El fichero routes.coffee contiene el mapeo entre las URLs que se visitan en el navegador y las acciones que hay que llevar a cabo por los controladores.

Los controladores son las clases que saben qué es lo que hay que hacer cuando se visita una URLs específica. Normalmente estas acciones consisten en la invocación de un modelo y la renderización de una vista con los datos del mismo.

Los modelos representan los datos de la aplicación. En el caso del simulador estos serán bloques, enlaces o eventos. Backbone hace una distinción entre una instancia de un único modelo (Model) y un conjunto de modelos del mismo tipo (Collection).

Por último, las vistas haciendo uso de plantillas HTML en las que se introduce la información de modelos o colecciones, se encargarán de incrustar estas plantillas en el contenido de la web. Las vistas son las que se encargan de detectar cualquier interacción del usuario y actuar frente a estas interacciones. Desde hacer click en un enlace, actualizar el contenido en función de los movimientos del ratón… cualquier cosa que implique modificar el contenido que un usuario ve o las interacciones con este último.

La aplicación además hace uso de imágenes, fuentes y otros tipos de contenidos estáticos. Estos se encuentran en la carpeta assets.

En la carpeta styles se encuentran los ficheros CSS con los que se consigue dar estilo a la aplicación.

Esta es la estructura sobre la que se ha construido la aplicación web que utiliza el simulador para realizar las visualizaciones de los datos. Siguiendo los patrones y convenciones de Backbone se ha conseguido desarrollar de forma muy rápida una aplicación vistosa, rápida y que facilita el uso del simulador más allá de una sencilla (y poderosa) terminal como se aprecia en la figura 33.

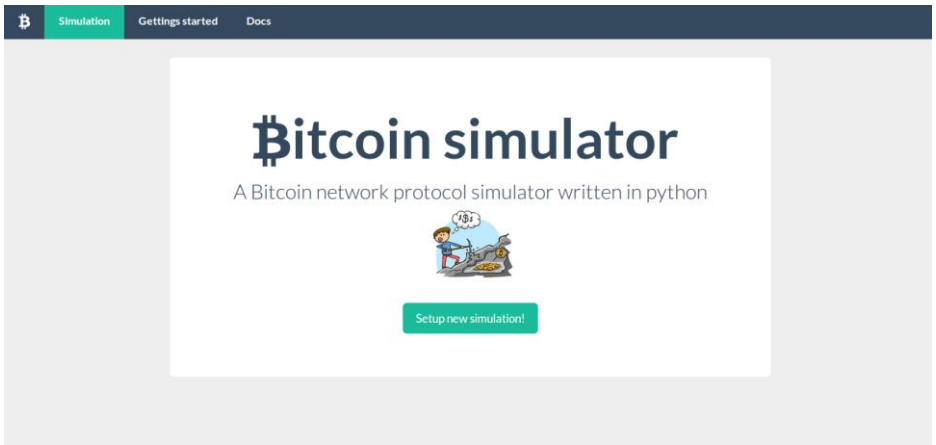

Fig. 33 Pantalla de bienvenida de la aplicación web.

6.5 Visualizaciones de datos

El último aspecto que queda por analizar de la aplicación web es cómo se han realizado las visualizaciones de los datos. Al comienzo del capítulo se comentaba que se había hecho uso

de la biblioteca D3.js. Esta biblioteca se basa en el uso de los datos para dirigir la creación y control de diferentes elementos gráficos dinámicos e interactivos. Está basada en el uso de estándares que siguen las doctrinas del W3C como es SVG. Presenta una API declarativa bastante expresiva centrada expresamente en el uso de los datos.

El principio de D3 son los datos. Permite hacer bindings de diferentes tipos de datos al DOM (Document Object Model) de una web para después aplicar transformaciones a este documento en función de los datos. Es importante destacar que no solo se utiliza D3 para crear visualizaciones, sino que se puede utilizar como se hace en la aplicación para la creación y manipulación de tablas. La gran expresividad de d3 permite hacer mucho en pocas líneas de código. En la aplicación web se han creado tres visualizaciones diferentes

Por un lado se creó un diagrama de la red Bitcoin con los nodos que la componen y sus interconexiones. El tamaño de un nodo y de un enlace entre ellos dependerá directamente de su capacidad de cómputo y del ancho de banda del enlace. Los nodos se pueden arrastrar para poder observar mejor sus conexiones y si se hace click en ellos, se destacarán sus conexiones sobre las demás de la red para facilitar su visualización y se mostrará un resumen de la información de ese nodo en la parte derecha de la pantalla, en la que también se muestra de forma permanente un resumen de la simulación con el total de días, nodos, enlaces, bloques y eventos que fueron generados. La potencia de D3 es tal que esta visualización apena ocupa 100 líneas de código y el resultado que se obtiene es más que satisfactorio. Las figuras 34 y 35 muestran dos de los ejemplos de gráficas construidas en la aplicación.

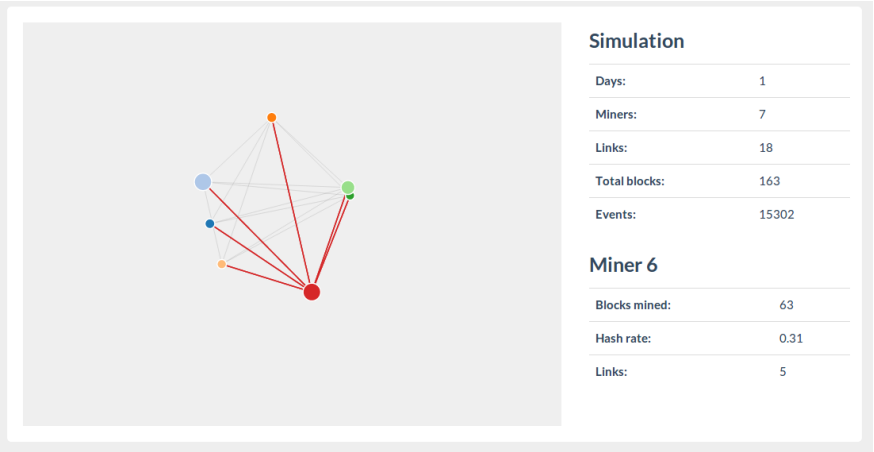

Fig. 34 Visualización de la red de Bitcoin.

La segunda visualización generada consiste en un diagrama de barras en el que se muestra el número de bloques que ha generado un nodo frente al total de bloques de la red. Además, se calcula el tiempo medio de generación de un bloque para poder comprobar si se mantiene el ratio de 1 bloque cada 10 minutos.

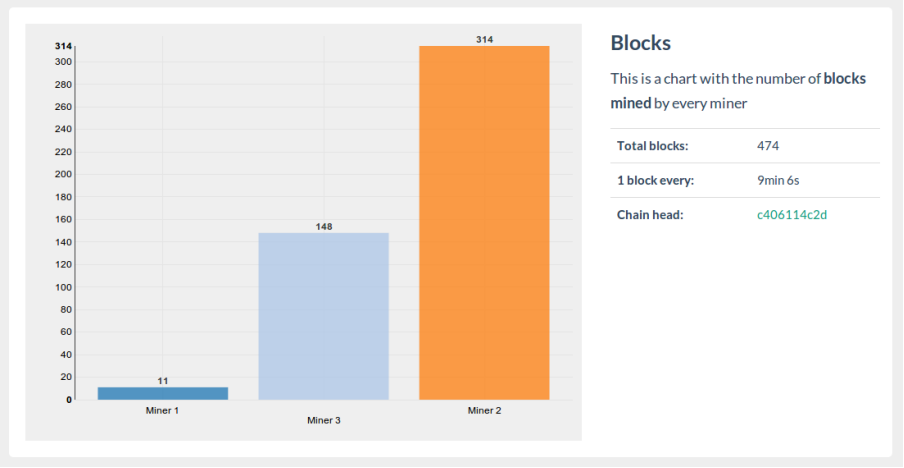

Fig. 35 Visualización de los bloques generados por cada nodo

Por último, se creó una tabla a través de la que se puede observar toda la cadena principal desde su último nodo hasta el bloque génesis, pudiendo observar su hash, su dificultar, el nodo que lo generó, su tamaño y el tiempo en que fue generado. Además, se permite buscar por el identificador de un bloque. Como en una simulación se pueden generar una gran cantidad de bloques, es necesario paginar estos resultados. Esta visualización se presenta en la figura 36.

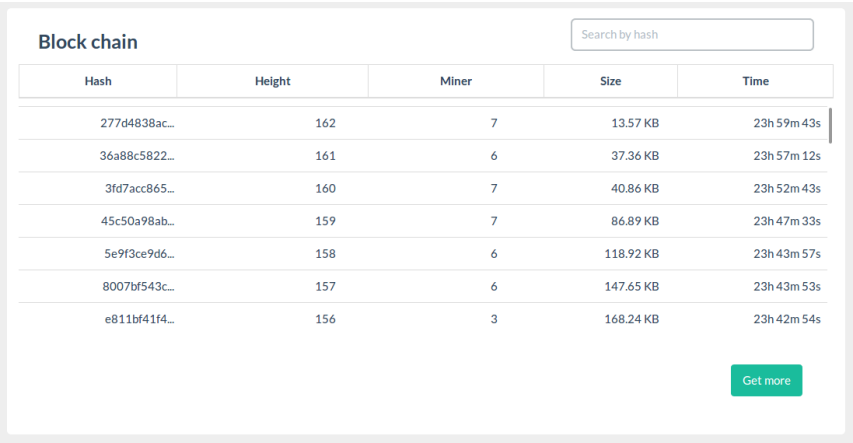

Fig. 36 Visualización de los bloques de la cadena principal

Con estas tres visualizaciones, al finalizar una simulación se puede obtener una idea de lo acontecido durante la ejecución de la misma. De una forma sencilla, visual y sin necesidad de estar divagando entre la gran cantidad de datos que se generan.

6.6 Ejecución remota del simulador

Como objetivo final de este proyecto se había marcado la implementación de algún mecanismo de control que permitiera lanzar las simulaciones desde la aplicación web. Este objetivo se ha logrado satisfactoriamente, de tal manera que desde la propia aplicación web se puede configurar la simulación que se va a lanzar, ejecutarla en un proceso en segundo plano en el servidor y cuando esta termine pasar a analizar los resultados.

Para poder implementar esta funcionalidad ha sido necesario hacer uso de Websockets, que permiten una conexión full-duplex entre el navegador y el servidor web con el objetivo de poder notificar la finalización de la simulación.

Además hacía falta algún tipo de proceso o biblioteca que se encargara de la ejecución del proceso de la simulación e informara de si finalización al servidor web para que este pudiera notificar a la aplicación web.

La comunicación entre estos dos procesos se puede realizar mediante el sistema de paso de mensajes que provee Redis. Lo único que hacía falta era la biblioteca encargada de ejecutar estos procesos en segundo plano. La biblioteca seleccionada fue Celery [63], una biblioteca para la ejecución de tareas de forma asíncrona basada en el envío de mensajes. Esta biblioteca tiene integración con Flask y permite usar Redis para el paso de mensajes y poder lanzar y conocer el estado de una tarea en todo momento.

En el fichero core.py del servidor web se define la configuración de celery, que se estará ejecutando en otro proceso en segundo plano de forma paralela al servidor web y que estará a la escucha de distintos canales de mensajes en Redis para comenzar las tareas que se hayan definido. Como se vio en el apartado anterior, las tareas se definen en el fichero tasks.py. En la figura 37 se muestra la inicialización de celery en flask.

```
# Method to create celery tasks
def make_celery(app):
    celery = Celery(app.import_name, broker=app.config['CELERY_BROKER_URL'])
    celery.conf.update(app.config)
    TaskBase = celery.Task
    class ContextTask(TaskBase):
        abstract = True
        def __call__(self, *args, **kwargs):
            with app.app_context():
                return TaskBase.__call__(self, *args, **kwargs)
    celery.Task = ContextTask
    return celery

celery = make_celery(app)
```

Fig. 37 Inicialización de celery

Este método es el que permite definir las tareas de Celery usando simplemente decorators de Python. Para poder lanzar una simulación desde la aplicación web es necesario crear un nuevo punto en la API que reciba los parámetros de la simulación y lance la tarea en segundo plano a través de Celery. Este código se muestra en la figura 38.

```
@app.route('/simulation', methods=['POST'])
@crossdomain(origin="*", headers=['Content-Type', 'Content-Disposition'])
def start_simulation():
    simulation = request.json
    logger.info("Starting %d days %s simulation with %d miners" %(simulation['days'], simulation['type'], simulation['miners']))
    # Start the simulation in the worker
    start_simulation_task.delay(simulation['miners'], simulation['days'], simulation['type'])
    # Return simulation parameters
    return jsonify(request.json)
```

Fig. 38 Ruta del servidor para iniciar una simulación

Cuando la simulación termina su ejecución, esta emite un mensaje a través de Redis en un canal en el que el servidor web estará escuchando. Cuando se reciba ese evento, el servidor, mediante los websockets difundirá un mensaje a todos los clientes conectados para indicar que una simulación acaba de terminar. El protocolo utilizado para el envío de mensajes a través de websockets es SocketIO [64].

En el fichero pubsub.py es donde se inicializan tanto los websockets como el proceso de escucha de los mensajes de Redis como se aprecia en la figura 39.

```python
def sub(pubsub):
    # Subscribe to channel
    pubsub.subscribe(app.config['SIMULATOR_NAMESPACE'])
    for message in pubsub.listen():
        # Process new messages when they arrive
        Greenlet.spawn(process_received_mesg, message)

def process_received_mesg(message):
    # Notify all clients the message
    logger.info("Received data from redis: " + repr(message['data']))
    socketio.emit('redis', message['data'], namespace=app.config['SIMULATOR_NAMESPACE'])

@socketio.on('connect', namespace=app.config['SIMULATOR_NAMESPACE'])
def connect():
    logger.info("Client connected")
    emit('status', {'data': 'Connected'})

pubsub = r.pubsub()
Greenlet.spawn(sub, pubsub)
```

Fig. 39 Inicialización de SocketIO

Por último, desde la aplicación web se define una interfaz desde la que configurar la simulación. Desde ella se pueden seleccionar parámetros como el número de nodos o el tiempo de ejecución. Esta pantalla se muestra en la figura 40.

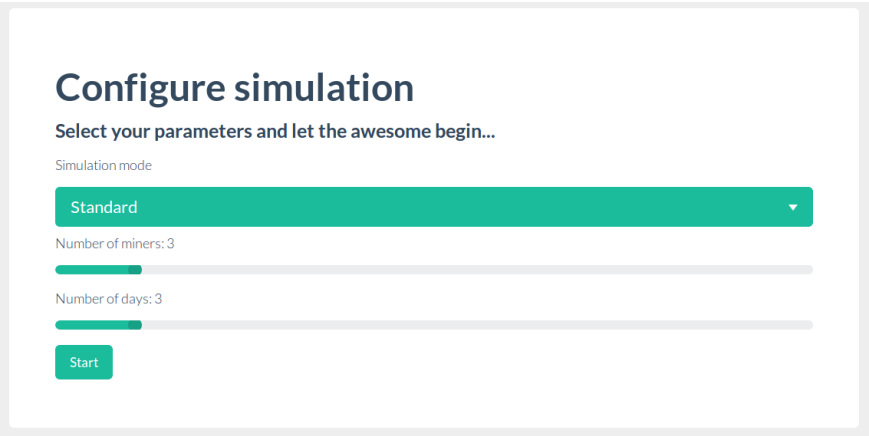

Fig. 40 Pantalla de configuración de la simulación

Una vez se lanza el simulador, se pasará a una pantalla de espera en la que se dará cierto feedback al usuario indicando el tiempo transcurrido desde el comienzo de la simulación como se observa en la figura 41.

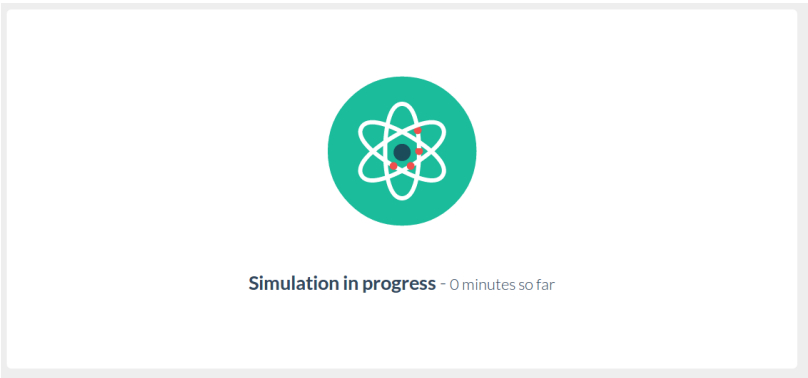

Fig. 41 Pantalla de carga de la simulación

Una vez termina la ejecución de la simulación, el usuario es redirigido a la página de estadísticas desde la que podrá analizar los resultados de la misma.

7 Ejecución del simulador

Una vez se había terminado el desarrollo del simulador, se podía empezar a jugar con él para ver si el desarrollo que se había realizado se correspondía con la realidad y los comportamientos esperados. Para ello, es necesario realizar pruebas con los diferentes modos implementados del simulador jugando con el tiempo de ejecución y el número de nodos.

En el simulador se definieron tres comportamientos diferentes de los nodos. En este capítulo se hará un análisis de los resultados devueltos por el simulador tras la ejecución de cada uno de estos modos.

7.1 Comportamiento natural

Este modo pretende simular el comportamiento habitual de la red, cuando todos los nodos se comportan siguiendo las reglas. Lo esperado al final de la ejecución es que todos los nodos acaben teniendo la misma cadena y que esta esté compuesta por bloques que han sido generados por todos los nodos. Habrá más bloques de aquellos nodos que tengan mayor capacidad de cómputo.

En todas las simulaciones que se llegaron a ejecutar con este comportamiento, independientemente de cómo estuvieran conectados los nodos y salvo que estuvieran aislados (algo que rara vez ocurría), todos los nodos acababan teniendo la misma cadena principal y en ella había algún bloque generado por los distintos nodos, a no ser que su capacidad de cómputo fuera muy pequeña, en cuyo caso podía ocurrir que el nodo no hubiera generado ningún bloque.

En cuanto al número de bloques generados, este siempre era próximo al ratio de 1 bloque cada 10 minutos, lo que es un buen indicador de que el simulador estaba teniendo comportamientos parecidos a la realidad. En la figura 42 hay un ejemplo de este comportamiento.

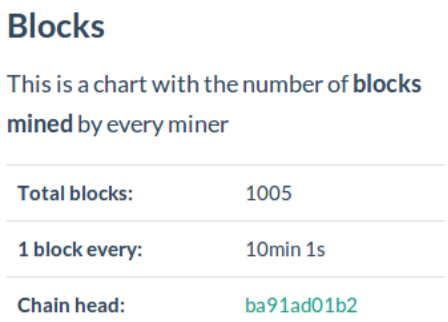

Fig. 42 Tiempo para la generación de un bloque en el simulador

Una característica que llama mucho la atención de las simulaciones es la gran cantidad de eventos de red que se generan en la misma. Parece lógico que este crezca de forma exponencial a medida que aumenta el número de nodos en la red y de forma lineal a medida que se aumenta el tiempo de la simulación, pero lo que extraña un poco es la cantidad en términos globales que se acaban generando y que se llegue a plantear la cuestión de si esto ocurre realmente así en la realidad. Una diferencia del simulador con la realidad es que aquí los bloques se solicitan y se envían de uno en uno, mientras que en los clientes Bitcoin reales se permite solicitar y enviar varios bloques en un mismo mensaje de red. En la figura 43 se presenta un resumen de una simulación.

Simulation

Days:	7
Miners:	9
Links:	25
Total blocks:	1005
Events:	186920

Fig. 43 Resumen de la ejecución de una simulación

7.2 Ataque del 51%.

La realización de este ataque permitiría a su autor realizar dobles gastos y bloquear ciertas transacciones. Como este comportamiento no se podía emular en el simulador, cuando se lleva a cabo este ataque, lo que se hace es que el atacante rechazará todos los bloques generados por otros nodos. Así, puesto que este atacante tiene más del 50% de la capacidad de cómputo de la red, al terminar la simulación es de esperar que todos los bloques de la cadena principal hayan sido generados por él.

Esto ocurría en la mayoría de las ocasiones, pero en algunas, la cadena principal presentaba un comportamiento raro al acercarse al final del tiempo. En estos casos, toda la cadena salvo los últimos bloques había sido generada por el atacante, pero al llegar a un determinado punto, el atacante dejaba de tener presencia en la cadena y comenzaban a hacerlo los demás nodos.

Las razones de este comportamiento son difíciles de intuir. Por un lado, podría ocurrir que llegado un punto el resto de la red genere bloques de forma más rápida que el nodo atacante, algo que es muy poco probable y que no explicaría por qué sólo ocurre este comportamiento al final de la cadena. Quizás sea más lógico pensar que este comportamiento tiene que ver más con el simulador en sí mismo, es decir, con Simpy y la gestión de los eventos cuando se acerca el final del tiempo, aunque no se ha podido terminar de determinar las razones del comportamiento. En la figura 44 se aprecia este comportamiento inesperado.

Hash	Height	Miner	Size	Time
94f4b23bf8...	72	9	58.22 KB	23h 45m 43s
a85ee683a1...	71	6	47.24 KB	23h 24m 56s
f7f10b7309...	70	1	42.90 KB	22h 57m 41s
fabbaeb25e...	69	9	35.00 KB	22h 43m 11s
6ff3160bca...	68	1	195.27 KB	22h 29m 58s
ce0cdce6d8...	67	10	107.42 KB	22h 13m 3s
999306517c...	66	10	109.19 KB	21h 55m 24s

Fig. 44 Cadena de una simulación del ataque del 51%

A pesar de esto, hay una cosa que está clara, el ataque tiene el comportamiento esperado. Al final de la ejecución todos los bloques de la cadena principal han sido generados por el atacante, lo que indica que este tiene potestad sobre la red y podría decidir qué transacciones realizar y cuáles no. Todas las recompensas por la generación de los bloques se las llevaría él.

7.3 Selfish attack

El análisis y comprensión del selfish attack ha sido un poco más complejo que el ataque anterior. Esto es debido a que por falta de tiempo no ha sido posible desarrollar visualizaciones que permitan observar las características propias de esta simulación, como es la cadena alternativa que va construyendo el atacante o los momentos en que el atacante hace pública esta cadena.

A pesar de no contar con las visualizaciones, se ha utilizado la API del servidor web para hacer diferentes consultas y poder intuir lo que estaba ocurriendo durante la simulación.

Las primeras ejecuciones de la misma, independientemente del número de días y nodos daban como resultado una cadena principal en la que no había ningún bloque del atacante. Este comportamiento parecía algo extraño, por lo que indagando un poco y observando la cadena alternativa del atacante, se veía como esta era construida de forma correcta, avanzando sobre la cabeza de la cadena de la red en un principio y aumentándola poco a poco con los bloques que va generando. Si la cadena principal se movía por encima de la cadena del atacante, entonces este la descartaba y comenzaba a trabajar en una nueva cadena alternativa. Si por el contrario, conseguía avanzar más que la cadena principal, la anunciaba para obtener un beneficio.

El procedimiento se ejecutaba correctamente y como se esperaba. Sin embargo, el ataque parecía no tener el éxito esperado. Esto es debido a las condiciones que se deben dar para que el ataque tenga éxito. Hay que tener en cuenta que el atacante sólo tiene un 30% de la capacidad de la red, no llegando a ser la mayoría y, para garantizar que cuando el atacante publique una cadena de la misma dificultad que la cadena principal, para que el resto de la red la acepte con seguridad, debería ponerse de acuerdo con ciertos nodos, que se verán atraídos por las promesas de mayor beneficio. Si esto no fuera así, para que la cadena del atacante fuera aceptada por el resto de la red, el otro 21% de la red necesario para llegar a la mayoría debería aceptarlo y esta elección en el simulador se hace por simplicidad eligiendo la cadena que llega primero, y esto dependerá de la velocidad y disposición de los enlaces para su aceptación. Por tanto, los resultados obtenidos no son tanta locura, pues el atacante no cuenta con el apoyo de otros nodos, sino que su victoria depende del azar, y las probabilidades de éxito se reducen muchísimo y es por eso que en muchas de las ejecuciones en la cadena principal aparecían muy pocos bloques de la cadena principal.

Concluyendo, se podría decir que la implementación del ataque es correcta pero que el simulador no está preparado para que pueda ser llevado a cabo al no tener los nodos implementado algún mecanismo que les permita dar prioridad a los bloques que provienen de un bloque determinado.

8 Conclusiones y trabajos futuros

En este trabajo se ha diseñado e implementado un simulador del protocolo Bitcoin completamente funcional capaz de simular los comportamientos más relevantes que se dan durante la ejecución del protocolo, permitiendo observar diferentes comportamientos de los nodos y la red en general en función de las acciones individuales de cada nodo. Se ha prestado especial interés en simular aquellos comportamientos que derivan en los ataques más importantes que se pueden llevar a cabo contra la red.

Para poder llevar a cabo este desarrollo ha sido necesario hacer un estudio exhaustivo del ecosistema Bitcoin, comenzando a un alto nivel para aprender los conceptos básicos y entender su funcionalidad, usos, ventajas y desventajas. Además de hacerse una idea del mundo Bitcoin desde el punto de vista del usuario, fue necesario hacer un estudio de los protocolos subyacentes que garantizan el correcto funcionamiento, la integridad y la seguridad del protocolo Bitcoin para poder garantizar que el comportamiento del simulador era lo más fiel a la realidad como se podía. Finalmente, también se hizo un análisis de los principales ataques se pueden llevar a cabo contra la red Bitcoin, el protocolo o los propios usuarios.

El proceso de creación del simulador comenzó con una fase de diseño en la que se decidieron qué elementos del protocolo estudiado eran necesarios implementar y la manera en que se implementarían, definiendo el alcance y funcionalidad que tendría el producto final desarrollado.

Como resultado de la fase de diseño, la fase de desarrollo fue dividida en 3 fases. Se comenzó con el desarrollo del simulador, que llevó la mayor parte del tiempo y durante el cual surgieron diferentes problemas que hubo que solventar. Terminado el simulador, dio comienzo la fase de desarrollo del servidor web que permitiría acceder a los datos de las simulaciones. El desarrollo concluiría con la implementación de una aplicación web desde la que se pueden observar distintas visualizaciones de los datos generados por la simulación y también realizar un control de la misma desde la propia aplicación web.

El desarrollo de este trabajo de final de master ha sido un proceso totalmente enriquecedor en múltiples aspectos. Se ha estudiado uno de los elementos del mundo actual que está marcando tendencia y que de aquí a un tiempo revolucionar (más de lo que ya está haciendo) muchos aspectos de la sociedad y el comercio electrónico. A su vez, se aprendió un nuevo lenguaje de programación con el que no se tenía ninguna experiencia previa, pero el aprendizaje no terminó ahí, sino que se adquirieron multitud de conocimientos relacionados con las redes, todos los detalles que las rodean y que hay que tener en cuenta a la hora de desarrollar un simulador. Ha sido una gran aventura en la que se ha disfrutado bastante por el camino.

Uno de los aspectos que más destacan de este trabajo es la innovación, pues en todo momento se ha trabajado con tecnologías de reciente creación sobre las que había poco escrito (Bitcoin tiene unos pocos años de vida y apenas había nada hecho sobre el simulador). Sin duda este trabajo es pionero en esta materia.

Una vez finalizado el trabajo, todavía quedan líneas abiertas para seguir trabajando:

- Mejorar el alcance del simulador para que pueda cubrir aspectos de más bajo nivel de los que cubre actualmente. Sería bastante interesante llegar a implementar la generación y validación de transacciones.
- Crear nuevas visualizaciones que puedan ofrecer más información acerca de lo ocurrido durante la simulación. En la versión actual sólo se utilizan unos pocos datos de los que se están almacenando. Hay mucha información para ser explotada.

- Permitir almacenar los resultados de una simulación a otra para poder realizar comparaciones entre distintas ejecuciones en las que se varíen los parámetros de configuración.
- Permitir la ejecución de varias simulaciones de forma simultánea. Actualmente sólo se permite ejecutar una única simulación. Sería interesante permitir que uno o varios usuarios de la aplicación lanzaran simulaciones al mismo tiempo.
- Añadir dinamismo a la red. En la versión actual del simulador, el número de nodos y su capacidad de computación permanece invariante a lo largo de la simulación. Permitir variaciones en los nodos o en su número aportaría mucho valor. Esto implicaría que un nodo tuviera la capacidad para descubrir otros nodos y pudiera gestionar las conexiones y determinar cuándo un nodo ya no está activo.

Bibliografía

[1] "Coinmap," [Online]. Available: http://coinmap.org/.

[2] "Ataque BTC China," [Online].

[3] S. Nakamoto, Bitcoin: A peer-to-peer electronic cash system.

[4] "Bitcoin Core," [Online]. Available: https://bitcoin.org/en/download .

[5] T. b. i. cirulation. [Online]. Available: https://blockchain.info/charts/total-bitcoins.

[6] "Bitcoin market price," [Online]. Available: https://blockchain.info/charts/market-price.

[7] "Number of Bitcoin transactions per day," [Online]. Available: https://blockchain.info/charts/n-transactions?showDataPoints=false&timespan=all&show_header=true&daysAverageString=7&scale=0&address=.

[8] "Bitcoin total number of transactions," [Online]. Available: https://blockchain.info/charts/n-transactions-total .

[9] "Paypal merchant fees," [Online]. Available: https://www.paypal.com/webapps/mpp/merchant-fees.

[10] "International Paypal restrictions," [Online]. Available: http://support.indiegogo.com/hc/en-us/articles/526426-International-PayPal-Restrictions.

[11] "Use bitcoins," [Online]. Available: http://usebitcoins.info/.

[12] "Bitcoin wallets," [Online]. Available: https://en.bitcoin.it/wiki/Wallet.

[13] "Bitcoin market capitalization," [Online]. Available: https://blockchain.info/charts/market-cap.

[14] "Bitcoin wallet," [Online]. Available: https://play.google.com/store/apps/details?id=de.schildbach.wallet.

[15] "QR codes," [Online]. Available: http://www.qrcode.com/en/about/.

[16] "NFC Forum," [Online]. Available: http://nfc-forum.org/what-is-nfc/.

[17] "Bitcoin mining hasdware comparison," [Online]. Available: https://en.bitcoin.it/wiki/Mining_hardware_comparison.

[18] "Pooled mining," [Online]. Available: https://en.bitcoin.it/wiki/Pooled_mining.

[19] "Bitcoin hashrate distribution," [Online]. Available: https://blockchain.info/pools.

[20] "How to buy bitcoins," [Online]. Available: http://howtobuybitcoins.info/.

[21] "LocalBitcoins," [Online]. Available: https://localbitcoins.com/.

[22] "Bitlegal Russia," [Online]. Available: http://bitlegal.io/nation/RU.php.

[23] "Now Vietnam quietly bans bitcoins," [Online]. Available: http://blog.bitlegal.io/post/78126277896/now-vietnam-quietly-bans-bitcoins.

[24] A. Greenberg, "Dark wallet is about to make Bitcoin money laundering easier than ever," 29 04 2014. [Online]. Available: http://www.wired.com/2014/04/dark-wallet/.

[25] R. Cohen, "Global Bitcoin computing power now 256 times faster that top 500 supercomputers, combined!," [Online]. Available: http://www.forbes.com/sites/reuvencohen/2013/11/28/global-bitcoin-computing-power-now-256-times-faster-than-top-500-supercomputers-combined/.

[26] "Securing your wallet," [Online]. Available: https://bitcoin.org/en/secure-your-wallet.

[27] "Bitcoin common vulnerabilites and exposures," [Online]. Available: https://en.bitcoin.it/wiki/Common_Vulnerabilities_and_Exposures.

[28] "Bitcoin race attack," [Online]. Available: https://en.bitcoin.it/wiki/Double-spending#Race_attack.

[29] J. R. Doucer, "The sybil Attack".

[30] "Bitcoin and Tor," [Online]. Available: https://en.bitcoin.it/wiki/Tor.

[31] I. Eyal and E. Gün Sirer, "Majority is not Enough: Bitcoin mining is vulnerable".

[32] "Bitcoin mining cartels: a total non-threat," [Online]. Available: http://web.archive.org/web/20110111043159/http://inertia.posterous.com/bitcoin-mining-cartels-a-total-non-threat.

[33] "Btcsim," [Online]. Available: https://github.com/rbrune/btcsim.

[34] "Simbit," [Online]. Available: https://github.com/ebfull/simbit.

[35] "Pynode," [Online]. Available: https://github.com/vivictormora/pynode.

[36] "python-bitcoinlib," [Online]. Available: https://github.com/jgarzik/python-bitcoinlib.

[37] "NS3," [Online]. Available: http://www.nsnam.org/.

[38] "NumPy," [Online]. Available: http://www.numpy.org/.

[39] "Erdos-Renyi model," [Online]. Available: https://en.wikipedia.org/wiki/Erd%C5%91s%E2%80%93R%C3%A9nyi_model.

[40] "Simpy," [Online]. Available: http://simpy.readthedocs.org/en/latest/.

[41] "Redis," [Online]. Available: http://redis.io/.

[42] "Top Github languges for 2013," [Online]. Available: http://adambard.com/blog/top-github-languages-for-2013-so-far/.

[43] "Websocket," [Online]. Available: http://www.websocket.org/.

[44] "PyCharm," [Online]. Available: http://www.jetbrains.com/pycharm/.

[45] "Git," [Online]. Available: http://git-scm.com/.

[46] "Repositorio del simulador en Github," [Online]. Available: https://github.com/vivictormora/btcsimulator.

[47] "Generadores de Python," [Online]. Available: https://docs.python.org/3/glossary.html#term-generator.

[48] "Simpy Issue Tracker," [Online]. Available: https://bitbucket.org/simpy/simpy/issue/51/multiple-consumers-of-filterstore-gets-not.

[49] "Python Package Index," [Online]. Available: https://pypi.python.org/pypi/pip.

[50] "Representational state transfer," [Online]. Available: https://en.wikipedia.org/wiki/Representational_state_transfer.

[51] "Flask," [Online]. Available: http://flask.pocoo.org/.

[52] "Sinatra," [Online]. Available: http://www.sinatrarb.com/.

[53] "Coffeescript," [Online]. Available: http://coffeescript.org/.

[54] "Brunch," [Online]. Available: http://brunch.io/.

[55] "Bower," [Online]. Available: http://bower.io/.

[56] "Backbone.js," [Online]. Available: http://backbonejs.org/.

[57] "Angularjs," [Online]. Available: http://angularjs.org/.

[58] "Chaplin.js," [Online]. Available: http://chaplinjs.org/.

[59] "D3.js," [Online]. Available: http://d3js.org/.

[60] "Bootstrap," [Online]. Available: http://getbootstrap.com/.

[61] "Flat-UI," [Online]. Available: https://designmodo.github.io/Flat-UI/.

[62] "Paradigma Convention Over Configuration," [Online]. Available: https://en.wikipedia.org/wiki/Convention_over_configuration.

[63] "Celery," [Online]. Available: http://www.celeryproject.org/.

[64] "Socket.IO," [Online]. Available: http://socket.io/.

[65] "ECDSA," [Online]. Available: https://en.wikipedia.org/wiki/Elliptic_Curve_DSA.

[66] "Algoritmo Hashcash," [Online]. Available: https://en.bitcoin.it/wiki/Hashcash.

[67] "Bitcoin protocol message types," [Online]. Available: https://en.bitcoin.it/wiki/Protocol_specification#Message_types.

[68] "Coinbase maturity of 100 transactions," [Online]. Available: https://bitcoin.stackexchange.com/questions/22548/coinbase-maturity-of-100-transactions.

Anexo A: El protocolo Bitcoin

Bitcoin funciona en una red de difusión para poder propagar todos los bloques y transacciones. Todas las comunicaciones se realizan sobre TCP y tiene soporte para IPv6. En este apartado se profundizará en la gestión de las conexiones, los diferentes mensajes existentes y su estructura así como la estructura de los principales elementos de Bitcoin, es decir, los bloques y las transacciones. Además, se explicará con más nivel de detalle el funcionamiento del propio protocolo entrando en aspectos como el minado, el cálculo de la dificultad y la validación de los bloques y las transacciones.

Hashes y árboles de Merkle

En el protocolo Bitcoin, cuando se calcula un hash, este es en realidad calculado dos veces. En la mayoría de las ocasiones, el algoritmo de hash utilizado es SHA256, aunque también hay soporte para RIPEMD160. En la figura 45 hay un ejemplo del proceso de doble hashing.

```
1  txt = Simulador de BTC
2  hash = 314bd2d1a9b6fb297a27d7fa470a41dbb7706fd987c96fa4a5b0e1326338a887 sha256(txt)
3  dhash = b1a83e30559aea06c404a25d38f62de3c2fa499c4209fb1791a9b064046b0d82 sha256(hash)
```

Fig. 45 Cálculo de un doble hash

Un árbol de Merkle consiste en un árbol binario de hashes (hashes dobles en el protocolo Bitcoin). Si en el momento de construir el árbol ocurriera que hay un número impar de nodos hoja, entonces el último elemento es duplicado para asegurarse de que la fila actual tiene un número par de hashes. Comenzando por la fila más profunda del árbol, se calcula el hash de cada uno de los elementos y por parejas son concatenados y se vuelve a calcular el hash de esa concatenación, reduciendo el número de nodos a la mitad. Este proceso se ejecuta recursivamente hasta que sólo queda un único nodo, que será la raíz del árbol. En la figura 46 se observa un ejemplo de este proceso.

```
d1 = dhash(a)
d2 = dhash(b)
d3 = dhash(c)
d4 = dhash(c)            # a, b, c are 3. that's an odd number, so we take the c twice

d5 = dhash(d1 concat d2)
d6 = dhash(d3 concat d4)

d7 = dhash(d5 concat d6)
```

Fig. 46 Cálculo de un árbol de Merkle

Firmas digitales

El protocolo Bitcoin utiliza el algoritmo ECDSA (Elliptic Curve Digital Signature Algorithm) [65] para firmar las transacciones sobre la curva secp256k1 que tiene la forma $y^2=x^3+ax+b$ siendo a=0 y b=7. Para poder representar una clave pública (que será un punto en la curva) en una única ristra de bytes se utiliza la codificación DER.

Direcciones Bitcoin

Una dirección (utilizada para enviar o recibir bitcoins) consiste en el hash de una clave pública con los siguientes parámetros. A continuación en la figura 47 se presenta un ejemplo de dirección Bitcoin.

```
Version = 1 byte of 0 (zero); on the test network, this is 1 byte of 111
Key hash = Version concatenated with RIPEMD-160(SHA-256(public key))
Checksum = 1st 4 bytes of SHA-256(SHA-256(Key hash))
Bitcoin Address = Base58Encode(Key hash concatenated with Checksum)
```

Fig. 47 Ejemplo de dirección Bitcoin

Estructura de un mensaje

Todos los enteros son codificados como little endian. La única excepción a esta regla son las direcciones IP y los números de puerto. En la tabla 2 se introduce la estructura de un mensaje en el protocolo Bitcoin.

Tamaño	Descripción	Tipo de datos	Comentario
4	Número mágico	Uint32_t	Número mágico que identifica la red de origen del mensaje
12	Comando	Char[12]	Cadena ASCII identificando el contenido del paquete
4	Longitud	Uint32_t	Longitud de la carga en bytes
4	Checksum	Uint32_t	Los primeros cuatro bytes de shar256(sha256(payload))
-	Payload	Uchar[]	Los datos reales

Tabla 2 Estructura de un mensaje del prtocolo Bitcoin

Entero de longitud variable

La codificación de estos enteros puede variar dependiendo del valor representado con el objetivo de ahorrar espacio. Los enteros de longitud variable siempre son precedidos por un array de un tipo de datos que puede variar en tamaño. Los números más largos son codificados usando little endian. En la tabla 3 se muestra la estructura de un entero de longitud variable.

Valor	Longitud	Formato
< 0xfd	1	Uint8_t
<= 0xffff	3	0xd seguido de la longitud como uint16_t
<= 0xffffffff	5	0fe seguido de la longitud como un unint32_t
-	9	0xff seguido de la longitud como un uint64_t

Tabla 3 Entero de longitud variable

Cadenas de longitud variable

Una cadena de longitud variable se representa utilizando un entero de longitud variable seguido de la propia cadena. En la tabla 4 se muestra la estructura de una cadena de longitud variable.

Tamaño	Descripción	Tipo	Comentario
-	Longitud	Entero de longitud variable	Longitud de la cadena
-	Cadena	Char[]	La propia cadena

Tabla 4 Cadena de longitud variable

Transacciones

Una transacción es un conjunto de datos firmados que se distribuye a través de la red y que se asocian en grupos denominados bloques. Normalmente en una transacción se hará

referencia a otras transacciones y en ella se envía una cantidad de bitcoins a una o varias claves públicas. Una transacción tiene el formato de la tabla 5.

Campo	Descripción	Tamaño
Versión	Actualmente 1	4 bytes
Contador de entrada	Entero positivo de longitud variable	1-9 bytes
Lista de entradas	Lista con las entradas de la transacción	Tantas entradas como indique el contador de entrada
Contador de salida	Entero positivo de longitud variable	1-9 bytes
Lista de salidas	Lista de salidas de la transacción	Tantas entradas como indique el contador de salida
Tiempo	Sello de tiempo que se establece cuando la transacción es definitiva	4 bytes

Tabla 5 Estructura de una transacción Bitcoin

Abstrayendo un poco esta estructura de datos, una transacción se caracteriza por tener una serie de entradas y salidas. A partir del ejemplo de transacción de la figura 48 se pasa a explicar su funcionamiento.

```
Input:
Previous tx: f5d8ee39a430901c91a5917b9f2dc19d6d1a0e9cea205b009ca73dd04470b9a6
Index: 0
scriptSig: 304502206e21798a42fae0e854281abd38bacd1aeed3ee3738d9e1446618c4571d10
90db022100e2ac980643b0b82c0e88ffdfec6b64e3e6ba35e7ba5fdd7d5d6cc8d25c6b241501

Output:
Value: 5000000000
scriptPubKey: OP_DUP OP_HASH160 404371705fa9bd789a2fcd52d2c580b65d35549d
OP_EQUALVERIFY OP_CHECKSIG
```

Fig. 48 Ejemplo de transacción Bitcoin

En esta transacción se están cogiendo 50 BTC de la salida 0 de la transacción f5d8… A continuación la salida envía estos 50 BTC a la dirección 4043... Cuando el destinatario de la transacción quiera gastar el dinero, tendrá que referenciar la salida número 0 de esta transacción como entrada de otra transacción.

Una entrada es una referencia a una salida de otra transacción. Como se puede ver, en una transacción se pueden listar como entrada. Los valores de cada una de estas entradas es sumado y esta cantidad es la que se puede usar en las salidas de una transacción. El campo "Previous Tx" es el hash de una transacción anterior en el tiempo. Index indica la salida específica de esa transacción y ScriptSig es una de las partes necesarias para poder reclamar el dinero de la transacción a través de un sistema de script que se introducirá a continuación.

Una salida de una transacción contiene las instrucciones para enviar los bitcoins a sus destinatarios. Value es el número de Satoshis (1 BTC = 100000000 Satoshi) que son enviados. ScriptPubKey es el segundo componente necesario para reclamar la salida de una transacción. Puede haber más de un salida. La suma del valor de las salidas ha de ser igual al de la entrada, puesto que para poder utilizar bitcoins estos tienen que ser referenciados como entrada de otra transacción. Si esto no es así, el dinero se perdería. En el caso de que la entrada de una transacción sean 50 BTC, pero sólo se quieran enviar 25 BTC, lo que se hace es crear dos salidas: una de 25 BTC al destinatario y otra de 25 BTC de vuelta al usuario que generó la transacción. Es lo que sería el cambio. Toda cantidad de dinero de una entrada que no se utilice en una salida se considera una comisión de la transacción y podrá ser reclamada por el usuario que genere el bloque en el que se incluya la transacción.

Para poder verificar que se puede recoger el dinero de las salidas referenciadas en las transacciones de entrada se utiliza un sistema de scripting que evalúa los valores de los campos ScriptSig y ScriptPubKey. Sólo se podrá utilizar la salida de una entrada si al terminar la ejecución del script, ScriptPubKey devuelve true.

Mediante el uso de este sistema, el usuario que genera una transacción puede crear condiciones bastante complejas que un usuario tiene que satisfacer para poder reclamar el dinero de la salida de esa transacción. Así, se puede crear una transacción que pueda ser reclamada por cualquier usuario o incluso se puede crear una transacción que no pueda ser reclamada por nadie.

Este sistema se basa en el uso de una pila en la que se procesa las instrucciones de izquierda a derecha. Una transacción se considerará válida si al final de la ejecución la pila contiene true. El emisor de la transacción emite las operaciones necesarias para reclamar la transacción y cualquiera que quiera utilizar su salida deberá proveer los valores a estas operaciones para que el resultado de su ejecución sea satisfactorio. Al comienzo del script se combinan los valores de ScriptSig y ScriptPubKey.

Se definen una serie de comandos y funciones que se pueden utilizar para componer un script. Hay constantes, operaciones clásicas de una pila (push, pop, swap,...), operaciones de control de flujo, operaciones lógicas y aritméticas además de una serie de comandos relacionados con la criptografía como es el cálculo de hashes y la comprobación de firmas digitales. En la tabla 6 se puede observar un ejemplo del proceso de verificación.

Pila	Script	Descripción
Vacía	<sig> <pubKey> OP_CHECKSIG	Se combinan los valores de scriptSig y scriptPubKey
<sig> <pubKey>	OP_CHECKSIG	Se introducen las constantes en la pila
true	Vacío	Se realiza la verificación de firma para los dos elementos de la cabeza de la pila

Tabla 6 Ejemplo del sistema de Script

Una vez entendido el funcionamiento de una transacción, se presenta la estructura de una transacción de entrada y de una transacción de salida en las tablas 7 y 8.

Campo	Descripción	Tamaño
Hash de la transacción anterior	Doble hash de una transacción pasada de la que utilizar sus	32 bytes
Referencia a la salida de la transacción anterior	Número no negativo que referencia una de las salidas de la misma	4 bytes
Longitud de scriptSig	Entero positivo de longitud variable	1-9 bytes
scriptSig	Script	Tantos bytes como se indique en el campo de longitud
Secuencia	-	4 bytes

Tabla 7 Estructura de una transacción de entrada

Campo	Descripción	Tamaño
Valor	Cantidad a ser transferida	8 bytes
Longitud de scriptPubKey	Entero positivo de longitud variable	1-9 bytes
scriptPubKey	Script	Tantos bytes como indique el campo de longitud

Tabla 8 Estructura de una transacción de salida

Bloques y el proceso de minado

Un bloque es una estructura de datos que contiene una serie de transacciones. Los bloques se organizan en una estructura de cadena, contando cada con una referencia al bloque anterior. Para que un bloque sea considerado válido y pueda ser añadido a la cadena de bloques, tiene que tener una solución a un puzle matemático. El proceso de minado es el procedimiento por el que un nodo de la red intentará resolver este problema, que no tiene una solución trivial. Sin embargo, una vez se tiene una solución es muy fácil saber si esta es válida o no. Este proceso de minado, como se veía en el al comienzo de este documento, es el que permite confirmar que un pago se ha realizado o no. Como recompensa a los usuarios que aportan su capacidad de cómputo para ayudar a la confirmación de transacciones, estos reciben una recompensa. Un bloque tiene la estructura de la tabla 9.

Campo	Descripción	Tamaño
Número mágico	Siempre es 0xD9B4BEF9	4 bytes
Tamaño	Bytes hasta el final del bloque	4 bytes
Cabecera	Compuesta por 6 elementos	80 bytes
Número de transacciones	Entero de longitud variable	1-9 bytes
Transacciones	Lista no vacía de transacciones	Tantos bytes como indique el contador

Tabla 9 Estructura de un bloque

La cabecera del bloque será estudiada más adelante en el proceso de minado.

Puesto que todas las transacciones son difundidas por toda la red, los nodos que se encuentren intentando resolver nuevos bloques, las recogerán y las añadirán al bloque que están intentando resolver.

Debido a que hay una recompensa para aquellos nodos que resuelvan un bloque, cada bloque contendrá la dirección Bitcoin a quien darle la recompensa. Este registro es conocido como coinbase y será siempre la primera transacción que aparezca en un bloque. El número de Bitcoins entregados a su descubridor se divide cada 210.000 bloques. Este tipo de transacción no hace uso de las salidas de otras transacciones, sino simplemente crea bitcoins de la nada.

Puesto que hay muchos nodos intentando simultáneamente obtener un bloque correcto, se puede dar el caso que la cadena de bloques en un momento determinado se divida por ejemplo si nos nodos dan al mismo tiempo con un bloque válido. La red será capaz de resolver este problema para que sólo una de las cadenas sobreviva.

Todo nodo Bitcoin aceptará como válida la cadena más larga. La longitud de una cadena no se refiere a la que más bloques tenga, sino que será aquella cuya dificultad combinada sea superior, es decir, que la dificultad combinada utilizada para resolver el puzle matemático de cada bloque sea mayor. Así se evita que un nodo genere una cadena de muchos bloques de poca dificultad y pueda llegar a ser considerada válida por la red.

Minado de bloques

A lo largo de este documento se ha definido el minado de bloques como el proceso por el que se validan las transacciones mediante la generación de bloques válidos y la adición de estos a la cadena principal. Para dar con un bloque válido es necesario dar con la solución a un problema cuya dificultad es establecida por la red en función del número de nodos y su capacidad de cómputo para mantener la generación de bloques a un ritmo constante.

Los bloques individuales para que se consideren válidos tienen que presentar una prueba de trabajo. Una prueba de trabajo es una serie de datos difícil de producir, pues su proceso de

generación puede ser costoso o consumir mucho tiempo y que tiene que satisfacer una serie de requisitos. El proceso de verificación de estos requisitos tiene que ser trivial. El proceso de producción de esta prueba puede ser un proceso aleatorio de baja probabilidad, de tal manera que se requieren bastantes ejecuciones en promedio antes de poder generar una prueba válida.

El objetivo de esto no es otro sino conseguir un consenso seguro y resistente a manipulaciones entre los nodos de la red Bitcoin. Este consenso consiste en determinar cuál es la cadena de bloques principal que contiene el histórico de transacciones y por el que se validan y confirman las mismas. A su vez, este proceso es el encargado de introducir nuevas monedas en circulación, consiguiendo así descentralizar este proceso y motivar a los usuarios para que colaboren en aportar seguridad al sistema.

El algoritmo de prueba de trabajo utilizado por Bitcoin se conoce como Hashcash [66]. Para que un bloque sea considerado válido el hash de su cabecera tiene que ser menor que un determinado valor objetivo, siendo este hash la prueba de trabajo. Puesto que cada bloque contiene en su cabecera el hash del bloque anterior en la cadena, se consigue que modificar un bloque dentro de la cadena implique modificar todos los bloques posteriores a él en la misma, haciendo que sea un proceso complejo, pues habría que obtener la prueba de trabajo para cada uno de estos bloques, consiguiendo así que la cadena de bloques sea muy difícil de alterar. Este algoritmo tiene una probabilidad de éxito bastante baja, haciendo que sea imposible predecir qué nodo de la red será el que genere el siguiente bloque. En la tabla 10 se puede apreciar la estructura de la cabecera.

Campo	Descripción	Actualizada cuando	Tamaño
Versión	Versión del bloque	Se actualiza el cliente	4
hashPrevBlock	Hash de la cabecera del bloque anterior	Se descubre un nuevo bloque	32
hashMerkleRoot	Hash de todas las transacciones del bloque	Se acepta una transacción	32
Time	Sello de tiempo	Cada varios segundos	4
Bits	Dificultad actual	Se ajusta la dificultad	4
Nonce	Número de 32 bits	Se prueba un nuevo hash	4

Tabla 10 Estructura de la cabecera de un bloque

La cabecera del bloque, entre otros parámetros contiene el árbol de Merkle de todas las transacciones del bloque, entre las que se encuentra la transacción coinbase, cuya dirección de destino será presumiblemente la del nodo que está tratando de encontrar una solución al bloque, con lo que se consigue que cada nodo al final esté hasheando un conjunto de datos único, es decir, que cada nodo tendrá un problema único al que encontrar una solución.

Mientras un nodo está buscando la solución a un bloque, este se puede ver alterado debido a un cambio en la dificultad, a que se haya generado un nuevo bloque o a que se haya confirmado una de las transacciones que contiene.

El campo objetivo (target) se trata de un número en punto flotante con un formato especial. Este campo en la mayoría de las ocasiones será el mismo para todos los usuarios. Por último el campo nonce es un contador que se va aumentando periódicamente cada vez que se prueba un nuevo hash. Si no fuera porque la dirección de la transacción coinbase es diferente para cada nodo, entonces prácticamente todos los nodos de la red estarían intentando dar con la solución al mismo problema y aquél con mayor capacidad de cómputo sería el que ganaría en todas las ocasiones. Así cada hash que un nodo calcula tiene la misma probabilidad de ser correcto que cualquier otro hash calculado por la red. La importancia de la capacidad de cómputo de un nodo reside en el número de hashes que es capaz de calcular por segundo.

Esto es debido a que el proceso de generación de un bloque no es un problema que se resuelva en función del tiempo dedicado, sino que es más bien una lotería. Cada hash dará un

número de 256 bits. Si este número es menor que el objetivo, entonces se gana y si no, se incrementa el nonce de la cabecera para probar otra vez.

Target y dificultad

El objetivo (target) se trata de un número en punto flotante de 256 bits compartido por toda la red. El hash de la cabecera de un bloque tiene que ser menor o igual que este número para que el bloque sea considerado válido. Mientras más pequeño es este número más difícil es obtener un bloque. El objetivo de la red es mantener la generación de bloques a un ritmo de 1 cada 10 minutos, por lo que cada 2016 bloques cada cliente Bitcoin comparará el tiempo real que se tardó en generar esos bloques con el valor objetivo de dos semanas (162414=2016) y modificará el valor de target en función del porcentaje de la diferencia. Esto hará la dificultad del problema más o menos difícil.

En todo momento se ha estado hablando de la dificultad para resolver el problema, pero no se ha definido hasta ahora. La dificultad es una medida que indica cuánto de difícil es obtener un hash por debajo de un target dado. Se calcula en la figura 49.

```
difficulty = difficulty_1_target / current_target
```

Fig. 49 Cálculo de la dificultad

difficulty_1_target representa un hash donde los 32 primeros bits son 0 y el resto 1. Normalmente en un bloque se almacena una versión comprimida de la dificultad para el target actual. Si el valor del target almacenado es 0x1b0404xb, entonces el valor hexadecimal del target será el de la figura 50.

```
0x0404cb * 2**(8*(0x1b - 3)) = 0x00000000000404CB000000000000000000000000000000000000000000000000
```

Fig. 50 Cálculo de la versión comprimida de la dificultad

La dificultad más alta (difficulty_1_target) se define como 0x1dffff, lo que da un valor hexadecimal equivalente al de la figura 51.

```
0x00ffff * 2**(8*(0x1d - 3)) = 0x00000000FFFF0000000000000000000000000000000000000000000000000000
```

Fig. 51 Cálculo de difficulty_1_target

En base a lo anterior, la dificultad para 0x1b0404cb es el ejemplo será la de la figura 52.

```
0x00000000FFFF0000000000000000000000000000000000000000000000000000 /
0x00000000000404CB000000000000000000000000000000000000000000000000
= 16307.420938523983 (bdiff)
```

Fig. 52 Cálculo de la dificultad de 0x1b0404cb

A partir de la dificultad actual se pueden calcular diferentes valores como la capacidad de cómputo de la red o incluso estimar el tiempo que se tardará en generar el próximo bloque.

La red Bitcoin

Dentro de la red Bitcoin, los diferentes nodos que la componen intercambian una serie de mensajes de red para difundir las transacciones y bloques y realizar otra serie de acciones propias de una red como la conexión y el descubrimiento de nodos. A continuación, sin entrar en detalle sobre su estructura, se presenta un listado de todos los mensajes que se pueden intercambiar entre los nodos [67].

- Version. Información acerca de la versión del programa. Cuando dos nodos se conectan no puede comunicar más información hasta que han intercambiado sus versiones.
- Verack. Mensaje enviado como respuesta a un mensaje version como muestra de que dos nodos tienen intención de conectarse.
- Addr. Devuelve un listado con las direcciones de los nodos conocidos en la red. Hay soporte tanto para IPv4 e IPv6.
- Inv. Permite a un nodo anunciar su conocimiento acerca de diferentes objetos (bloques o transacciones por ejemplo). Puede ser recibido sin que se haya solicitado o como respuesta al comando getblocks. Los datos que se envían es una lista y no todos los datos, sino parte que los identifica (el hash)
- Getdata. Este comando es la respuesta a inv y se utiliza para obtener el contenido de un objeto específico.
- Notfound. Mensaje de respuesta a getdata si los datos solicitados no pudieron ser encontrados o no estaban disponibles.
- Getblocks. Comando que devuelve un paquete inv con una lista de los bloques contenidos dentro de un rango enviado por parámetros.
- Getheaders. Solicitud de un listado de hashes de bloques contenidos en un rango específico.
- Tx. Describe una transacción Bitcoin y se envía como respuesta a un mensaje getdata.
- Block. Envío de un bloque como respuesta a un comando getdata.
- Headers. Respuesta al comando getheaders con un listado de bloques.
- Getaddr. Petición para obtener información acerca de los nodos conocidos de la red.
- Mempool. Petición para obtener el listado de transacciones que un nodo ha verificado pero todavía no han sido confirmadas. La respuesta será un mensaje de tipo inv.
- Ping. Comando utilizado para conocer si la conexión TCP/IP con un nodo sigue siendo válida.
- Pong. Mensaje de respuesta al mensaje ping. Este mensaje se genera a partir de un nonce enviado en el ping.
- Reject. Este mensaje se envía como respuesta a un mensaje que ha sido rechazado. Existen diferentes códigos de error para indicar las causas del rechazo.
- Alert. Permite enviar notificaciones generales por toda la red.

Interacciones en la red

En este apartado se comenta el proceso de conexión de un nodo a la red y la forma en que se retransmiten las transacciones y bloques además de explicar la forma en que se descubren nuevos nodos en la red y cómo se renuevan o terminan las conexiones con estos nodos.

Para conectarse con un nodo, lo primero que hay que hacer es enviar un mensaje version con la versión del cliente que se esté utilizando. El otro nodo responderá con un verack y la versión de su cliente sólo si acepta conexiones de la versión de origen. Acto seguido habrá que responder con un verack si se aceptan conexiones de la versión del nodo remoto.

A continuación se deben intercambiar mensajes getaddr y addr y almacenar todas las direcciones de las que no se tenga conocimiento.

A partir de ahora cuando un nodo envía una transacción, éste enviará un mensaje inv con ella en su interior a todos los nodos conocidos. Estos nodos solicitarán la información completa de la transacción con un mensaje getdata. Si estos consideran que la transacción es válida después de haberla recibido, entonces éstos la retransmitirán a sus nodos con un nuevo mensaje inv y así sucesivamente. Un nodo solicitará una transacción y la retransmitirá sólo si no tiene conocimiento previo de ella, por lo que nunca se retransmitirá una transacción de la que ya se tenga conocimiento, de tal manera que llegará un punto en que una transacción se "olvide" si no es añadida a ningún bloque después de un tiempo. Si esto ocurriera, entonces el emisor y receptor de la misma deberán volver a transmitirla.

Cualquier nodo que se encuentre minando deberá escuchar y recolectar aquellas transacciones válidas recibidas y trabajar para incluirlas en un bloque. Cuando algún nodo encuentre un nodo válido, entonces enviará un mensaje inv con él a todos los nodos conocidos siguiendo el procedimiento utilizado con las transacciones.

Todo nodo enviará un mensaje addr con su IP cada 24 horas. Este mensaje será retransmitido a otros nodos, consiguiendo que todo el mundo tenga una idea clara de qué IPs están conectadas a la red en todo momento.

Los mensajes de tipo alert son distribuidos a toda la red. No se utilizan mensaje inv, sino que si se recibe un mensaje alert válido, éste será retransmitido a todos los nodos.

Cuando un nodo se conecta a la red, éste enviará un mensaje getblocks con el hash del último bloque del que se tuvo conocimiento. Si los nodos que reciban este mensaje consideran que este no es el último bloque, responderán con un mensaje inv con hasta 500 bloques que se encuentren por encima del bloque listado. A continuación se deberá solicitar cada uno de estos bloques con un getdata. Se debe realizar este proceso de forma iterativo hasta que se hayan obtenido todos los bloques.

A la hora de decidir con qué nodos conectarse, se ordena la base de datos de direcciones por tiempo de última conexión, aunque añadiendo cierta aleatoriedad. El proceso de descubrimiento de nuevos nodos se puede conseguir mediante mensajes addr, mediante el uso de servidores DNS o mediante el uso del protocolo IRC, aunque este último mecanismo no está activado por defecto.

Si pasa un tiempo de 30 minutos desde que un nodo envió algún mensaje, entonces enviará un latido (heartbeat) para mantener la conexión con sus nodos activa. Si han pasado más de 90 minutos desde que un nodo envió algún tipo de mensaje, entonces se dará por cerrada la conexión con ese nodo.

Validación de bloques y transacciones

A lo largo de este apartado se ha contado que un nodo siempre tendrá en su posesión un listado o inventario de transacciones y nodos, siendo estos identificados de forma unívoca mediante si hash o el hash de su cabecera respectivamente. De forma conceptual, un nodo tendrá en sus diferentes listados de transacciones y de bloques:

Transaction pool. Colección no ordenada de transacciones que no se encuentran en algún bloque de la cadena principal pero para la que se tiene información de todas sus transacciones de entrada.

Orphan transactions. Conjunto de transacciones que no puede formar parte del pool ya que no se tiene información de todas sus transacciones de entrada.

Bloques en la cadena principal. Las transacciones que se encuentren en algún bloque de esta colección pueden ser consideradas (hasta cierto punto) como aceptadas.

Bloques en alguna rama secundaria. Conjunto de bloques que se encuentran en algún fork de la cadena principal.

Orphan blocks. Conjunto de bloques que no encajan en la cadena principal debido a que falta información de algún predecesor en la cadena.

Los bloques de las dos primeras colecciones forman un árbol cuya raíz es lo que se conoce como el “genesis block”, que fue el primer bloque en la cadena de bloques y por el que se generaron los primeros 50 bitcoins. La rama principal será la más larga, es decir, aquella con la mayor dificultad sumando las dificultades individuales de cada bloque.

El proceso para validar transacciones y bloques se trata de un algoritmo determinista constituido por una serie de pasos. Cuando un nodo recibe un mensaje tx que contiene una transacción, procederá de la siguiente manera:

4. Comprobar sintaxis.
5. Comprobar que la lista de entrada o de salida no está vacía.
6. Comprobar que su tamaño es menor que el tamaño máximo de un bloque.
7. Cada valor de una salida, así como el valor total debe ser un valor legal.
8. Asegurarse que ninguna de las entradas es una coinbase.
9. Comprobar que el tamaño de la transacción es de al menos 100 bytes, que el campo nLockTime no pasa de 31 bits y que el número máximo de operaciones de firma en los scripts no es mayor de 2.
10. Rechazar las transacciones que no sean estándares.
11. Rechazar si ya se posee la transacción en el pool de transacciones o si se encuentra en algún bloque de la cadena principal.
12. Para cada entrada, si la salida referenciada ya se encuentra en otra transacción en el pool, rechazar la transacción.
13. Para cada entrada, buscar en la cadena principal y en el pool la salida referenciada. Si no se encuentra para alguna de las entradas, mover la transacción al pool de transacciones huérfanas.
14. Para cada entrada, si la salida es de tipo coinbase, esta ha de tener una madurez de 100, es decir, haber sido confirmada 100 veces, o en otras palabras, que el bloque en el que se encuentra dentro de la cadena principal esté a 100 bloques de la cabeza de la misma. Esto se realiza para evitar tener que volver a crear una nueva transacción si la actual terminase en un bloque huérfano [68].
15. Para cada entrada, si la salida referenciada no existe, rechazar la transacción.
16. Utilizando las salidas referenciadas por cada entrada, comprobar que cada valor de entrada y la suma total es válida.
17. Rechazar la transacción si la suma de las entradas es menor que la suma de las salidas.
18. Rechazar la transacción si la comisión de la transacción (suma del valor de las entradas menos el valor de las salidas) es demasiado bajo como para introducirla en un bloque vacío.
19. Comprobar que el valor scriptPubKey de cada salida acepta los valores de las entradas. Rechazar en caso contrario.
20. Añadir la transacción al pool de transacciones.
21. Añadir la transacción a la cartera (wallet) si pertenece al usuario del nodo.
22. Retransmitir la transacción al resto de nodos.
23. Para cada transacción dentro del pool de transacciones huérfanas, si alguna de ellas utiliza la transacción actual como alguna de sus entradas, ejecutar este algoritmo con ella.

Cuando un nodo recibe un mensaje block que contiene la información de un bloque, ejecutará el siguiente procedimiento:

1. Comprobar sintaxis.
2. Rechazar si ya se tiene el bloque en alguno de los tres listados posibles.
3. La lista de transacciones no puede ser vacía.
4. El hash del bloque debe satisfacer las condiciones del target actual y de la prueba de trabajo.
5. El sello de tiempo del bloque no puede ser mayor de dos horas en el futuro.
6. La primera transacción del bloque debe ser coinbase, las demás no podrán serlo.

7. Para cada transacción del bloque, se deberán llevar a cabo los puntos 2-4 del algoritmo anterior.
8. La longitud de scriptSig de la primera transacción debe estar entre 1 y 100.
9. Rechazar si la suma de las operaciones de firma de todas las transacciones es mayor que la constante MAX_BLOCK_SIGOPS.
10. Verify el árbol de Merkle del bloque.
11. Comprobar si el bloque anterior se encuentra en la cadena principal o en alguna de las cadenas secundarias. Si no, añadir el bloque al a los huérfanos y solicitar al nodo del que se recibió el bloque el bloque anterior.
12. Comprobar que el valor de nBits coincide con el valor de la dificultad actual.
13. Rechazar si el sello de tiempo es la media de los últimos 11 bloques o anterior.
14. Para algunos bloques antiguos (cuando se descarga la cadena por primera vez) comprobar que su hash coincide con algunos valores ya conocidos.
15. Añadir el bloque a una cadena. Puede haber tres posibilidades. 1. El bloque extiende la cadena principal. 2. El bloque extiende una cadena que no es la principal, pero no añade suficiente dificultad como para que se convierta en la nueva cadena principal. 3. El bloque da lugar a una nueva cadena que se convierte en la cadena principal.
16. En el caso 1.
    1. Para cada una de las transacciones excepto la coinbase:
        1. Para cada entrada, buscar en la rama principal la salida referenciada. Rechazar si no se encuentra alguna de las salidas.
        2. Para cada entrada, si esta referencia la salida n de la transacción anterior pero esta tiene menos de n+1 salidas, rechazar.
        3. Para cada entrada, si la salida referenciada es coinbase, esta deberá tener una madurez de 100.
        4. Comprobar las firmas de cada entrada.
        5. Para cada entrada, si la salida referenciada ya ha sido utilizada por otra transacción en la rama principal, rechazar el bloque.
        6. Comprobar que el valor de cada salida referenciada por las entradas así como su suma, se encuentran dentro de un valor legal.
        7. Rechazar si la suma de las entradas es menor que la suma del valor de las salidas.
    2. Rechazar si el valor de la coinbase es mayor que la suma de la recompensa y las comisiones por transacción.
    3. Añadir al wallet aquellas transacciones que sean del usuario.
    4. Borrar del pool de transacciones aquellas que se encuentren en el bloque.
    5. Retransmitir el bloque a los demás nodos.
    6. Si el bloque fue rechazado, éste no formará parte de la cadena principal.
17. Para el caso 2, no hay que hacer nada.
18. Para el caso 3.
    1. Buscar el bloque de la cadena principal en el que se produce la separación.
    2. Redefinir la rama principal para que su cabeza sea el bloque que produce la separación.
    3. Para cada bloque de la cadena secundaria (ahora principal) desde el bloque de la separación hasta la cabeza.
        1. Realizar las comprobaciones 3-11.
        2. Para cada transacción excepto la coinbase.
            1. Para cada entrada, buscar en la rama principal la salida referenciada. Rechazar si no se encuentra alguna de las salidas.

2. Para cada entrada, si esta referencia la salida n de la transacción anterior pero esta tiene menos de n+1 salidas, rechazar.
3. Para cada entrada, si la salida referenciada es coinbase, esta deberá tener una madurez de 100.
4. Comprobar las firmas de cada entrada.
5. Para cada entrada, si la salida referenciada ya ha sido utilizada por otra transacción en la rama principal, rechazar el bloque.
6. Comprobar que el valor de cada salida referenciada por las entradas así como su suma, se encuentran dentro de un valor legal.
7. Rechazar si la suma de las entradas es menor que la suma del valor de las salidas.

3. Rechazar si el valor de la coinbase es mayor que la suma de la recompensa y las comisiones por transacción.
4. Añadir al wallet aquellas transacciones que sean del usuario.

4. Si se rechaza en algún punto, restablecer la rama principal a su estado previo a la validación. Parar.
5. Para cada bloque en la antigua cadena principal, desde la cabeza hasta el bloque de la separación.

1. Para cada transacción no coinbase del bloque.

1. Aplicar las comprobaciones 2-9 (salvo el 8) de una transacción.
2. Añadir al pool de transacciones si fue aceptada. En otro caso seguir con la siguiente transacción.

6. Para cada bloque de la nueva cadena principal (desde la cabeza hasta el bloque de la separación).

1. Eliminar del pool de transacciones aquellas transacciones que se encuentren en el bloque.

7. Retransmitir el bloque a los demás nodos.

19. Para cada bloque huérfano cuyo bloque anterior sea el actual ejecutar este algoritmo con él.