Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Análisis de estabilidad y seguridad en una red de *Proof of Stake*: el ejemplo de Algorand

Tesis de Licenciatura en Ciencias de la Computación

Ezequiel Vera

Director: Esteban Mocskos

Buenos Aires, 2023

ABSTRACT

En este trabajo de tesis estudiamos `Algorand`, una red de *blockchain* basada en *Proof of Stake*, analizando la estabilidad y seguridad de la misma en dos etapas.

En una primera etapa, experimentamos con la estabilidad y adaptación a cambios en la topología física de la red. Para ello creamos experimentos con distintas configuraciones en los que afectamos enlaces agregando latencias de manera incremental. En todos ellos logramos degradar y hasta detener la red al afectar un 20 % del *stake* participante del consenso, encontrando un potencial vector de ataque.

Como segunda etapa, ante la factibilidad de un ataque de este tipo, nos interesó poder reconstruir la topología lógica de la red para ubicar a los nodos de mayor *stake*, a partir de los mensajes recibidos por un *relay* del sistema. Experimentamos con redes de diferentes tamaños hasta replicar la escala de la red real. Para cada uno creamos un escenario en el que el *stake* se distribuye de manera homogénea entre los nodos, y otro heterogéneo en el que se respeta la distribución real de la red para fines de 2022.

Para los escenarios homogéneos, logramos predicciones que aciertan por completo en las conexiones de los nodos a los *relays* vecinos del *relay* analizado, pero que comienzan a fallar en la escala realista (bajando el nivel de detección y generando falsos positivos).

Para los escenarios heterogéneos observamos un comportamiento similar: partimos de predicciones exactas para redes más pequeñas, que comienzan a fallar a medida que incrementamos el tamaño de la red. Sin embargo, nuestro algoritmo pudo predecir en escala realista de manera exacta y precisa las conexiones de las cuentas más importantes de la red (aquellas con mayor *stake*).

Este estudio es el primero para la red de Algorand en base a los aspectos analizados, con emulaciones de la misma en escala realista.

Palabras claves: Blockchain, `Algorand`, consenso, emulación, `SherlockFog`, topología, *Proof of Stake*.

Índice general

1. INTRODUCCIÓN

1.1. Motivación

En los últimos años, se vivió un auge en el mundo de las criptomonedas y tecnologías *Blockchain* que trajo consigo nuevos temas de discusión sobre su utilización y costo de funcionamiento. Se plantea un problema en común a todas las opciones tecnológicas que fueron apareciendo en estos años: cómo hacer para que una red conformada por nodos (generalmente) descentralizados se ponga de acuerdo en el estado en que se encuentra el sistema sin permitir que un usuario malicioso, o un número *pequeño* de ellos, pueda afectar su funcionamiento incorporando transacciones o información falsa. Este problema es conocido como el **problema de los generales bizantinos**. Cada red propone un método de enfrentar este problema, ya sea asociando un costo en recursos computacionales a la incorporación de información (como es el caso de Bitcoin) o relacionando la credibilidad que tiene un actor de la red a una cantidad de dinero que tiene invertido en la red (Ethereum 2.0, Solana, `Algorand`).

Pero así como solucionan, en diferente medida, el problema del acuerdo también traen consigo nuevos problemas o factores a considerar: en los últimos años hubo un crecimiento enorme en la cantidad de personas que participan del minado en redes como Bitcoin que requieren un esfuerzo computacional y, por ende, energético grande como parte de su consenso de *Proof of Work*. La figura 1.1 presenta la cantidad de recursos computacionales (*hash power* o *hashrate*) utilizado en Bitcoin desde su aparición en 2008 hasta la actualidad. Se puede ver el enorme incremento de recursos invertidos por los mineros de tal manera que los utilizados antes de 2017 no llegan a percibirse por la escala.

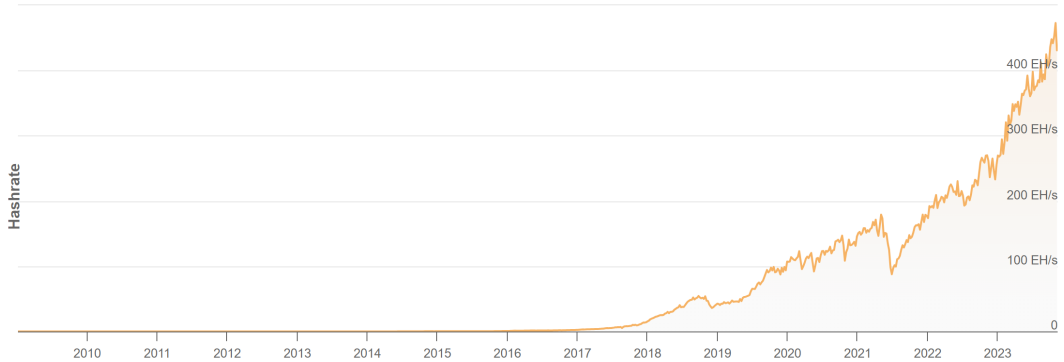


Fig. 1.1: Hashrate de Bitcoin desde sus inicios hasta la actualidad, expresado en ExaHashes (10e17 hashes) por segundo [1].

Este incremento en la cantidad de cómputo realizado por usuarios para participar de la red conlleva un aumento lógico del consumo energético total de la red. La figura 1.2 presenta el consumo energético de Bitcoin para cada año y el acumulado. Al igual que ocurría con los recursos computacionales, se nota un incremento creciente en el consumo energético, pero se nota uno de gran magnitud entre 2017 y 2018.

Si bien en algunos casos puede resultar redituable la inversión de recursos en el minado de Bitcoin, se genera una controversia con respecto al gasto energético de la red por todo

el esfuerzo computacional que se desecha y el impacto que tiene esto, no solo a nivel del malgasto de energía, sino en la contaminación ambiental que conlleva [2]. Por ejemplo, tomando datos de distintas fuentes de consumo energético, en la figura 1.3 se muestra cómo Bitcoin ya supera el consumo anual de todos los televisores o heladeras de Estados Unidos, alcanzando casi el consumo mundial de la industria del cobre.

En base a estos problemas, surgieron otros mecanismos de consenso que evitan cálculos artificiales y, por ende, el desperdicio de energía. Entre los más importantes se encuentra el de *Proof of Stake (PoS)*, donde cada actor de la red participa de las decisiones votando con el peso de su participación (o *stake*). Sin embargo, si bien solucionan aspectos negativos del protocolo anterior, introduce nuevos desafíos.

En este trabajo tomamos como motivación algunos de ellos, como la toma de decisiones ante alteraciones en la topología física o cómo gestiona la red la anonimidad de las cuentas que determinan el avance de la misma, evitando potenciales ataques. Para ello, trabajaremos con la red de `Algorand`, cuyo protocolo de consenso se basa en *Proof of Stake*.

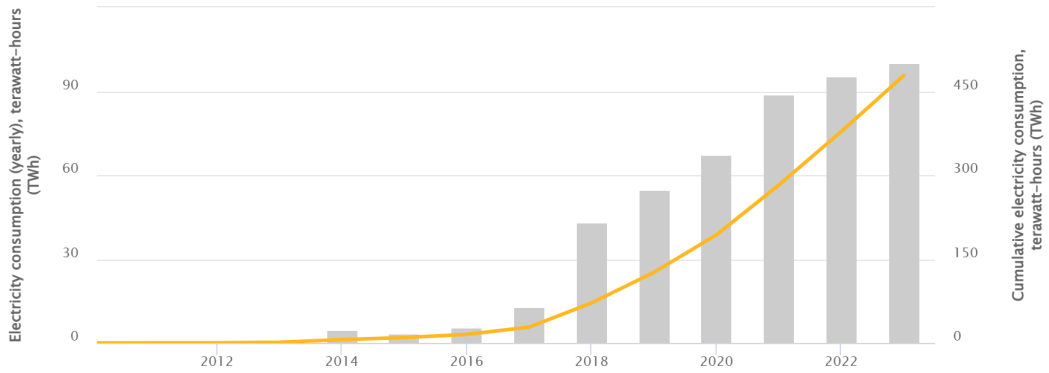


Fig. 1.2: Consumo de energía anual y acumulado por la red de Bitcoin [3].

1.2. Conceptos iniciales

En el desarrollo de este trabajo analizaremos distintos aspectos de la red de `Algorand` que implican la comprensión del funcionamiento y diferentes propiedades que caracterizan a un sistema basado en *Blockchain*. Estos conceptos generan la base teórica que luego nos permite presentar de manera detallada cómo `Algorand` resuelve e innova con respecto a otras redes.

Vamos a revisar estos conceptos iniciales sin detenernos en detalles de implementación, pero dejando las referencias adecuadas para que el lector interesado pueda luego profundizar en el tema.

1.2.1. Funciones de hash

Uno de los bloques primordiales para poder construir una *blockchain* son las funciones de hash. Una función de hash es una función tal que dada una entrada de tamaño variable, convierte la misma en una salida con un largo fijo predeterminado (llamado directamente *hash*). Una buena función de hash debe cumplir con las siguientes características:

1. Para la misma entrada, el resultado debe ser siempre el mismo. Es decir, es **determinística**.

2. Aún para entradas cuasi-idénticas, la salida no debe mostrar una **relación aparente**.

3. Dada una salida, debe ser computacionalmente **muy costoso** (podríamos decir *imposible* con un exceso de optimismo) calcular el valor de la entrada que la generó (se dice que *es de una vía* o que tiene un *único sentido*).

4. Para cualquier par de entradas, las salidas puede ser iguales solo con una muy baja probabilidad, es decir debe haber pocas **colisiones**. Está claro que esto depende también de los conjuntos imagen y dominio usados para la función.

5. Calcular el resultado de la función debe ser rápido y razonablemente *barato* en términos computacionales.

Si bien el punto 1 aplica para todas las funciones de hash, los demás puntos la convierten en una función apta para aplicaciones criptográficas. Dependiendo del nivel de complejidad y solidez del algoritmo utilizado para esta función, se cumplirán los puntos 3 y 4, entendiendo que la complejidad computacional es algo que depende del algoritmo en sí. Sin embargo, a medida que avanza el desarrollo del *hardware*, existen implementaciones de estas funciones de hash que en algún momento fueron seguras, pero que con las mejoras

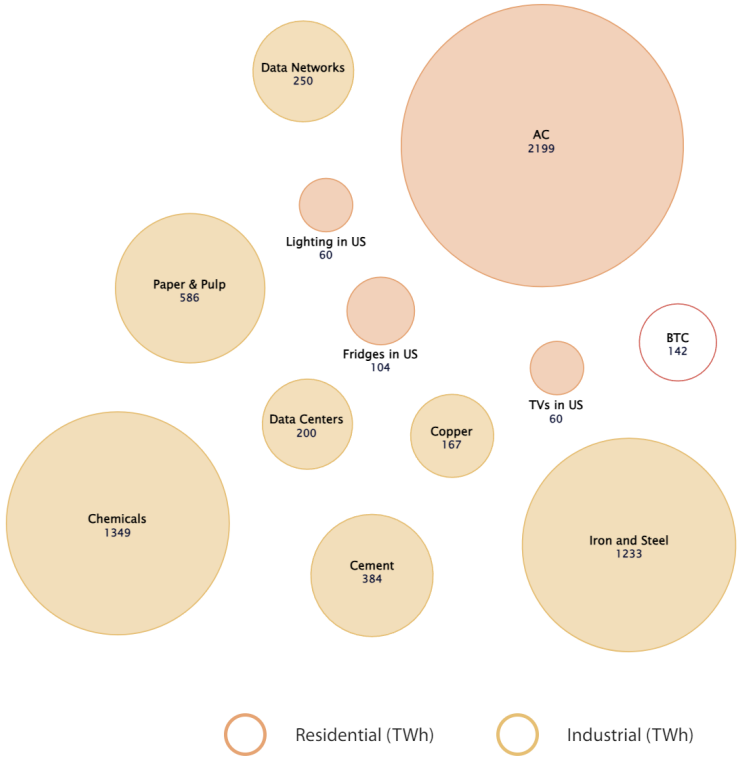


Fig. 1.3: Comparación de consumo actual de energía por hora (en terawatt-hora) de la red de Bitcoin con otras fuentes de consumo, en base a datos de 2018 y 2019 ajustados al año 2022 [3].

computacionales se lograron *romper*: es decir, se encontraron maneras de calcular la entrada de la función para una salida dada, sea por fuerza bruta o por alguna característica explotable del algoritmo.

La idea principal detrás de una función de hash es que sea fácil de computarla a partir de una entrada, pero que obtener una entrada a partir de su resultado solo se pueda hacer por medio de una búsqueda exhaustiva. Es decir que para poder obtener el valor que generó una salida, el único modo disponible sea probar con diferentes entradas hasta encontrar la que genera la salida buscada. No debería haber *pistas* para optimizar esta búsqueda que puedan guiar otra estrategia más *astuta*.

1.2.2. Claves asimétricas: públicas y privadas

Otro elemento importante en la criptografía, y que es otra base para los sistemas basados en *Blockchain*, son las claves asimétricas. Las claves asimétricas se componen de dos claves diferentes: una clave pública y otra privada que se generan en conjunto, con la característica única de que todos los mensajes cifrados por la clave privada pueden ser descifrados sólo por la clave pública, y viceversa.

La clave pública puede ser compartida libremente, mientras que la privada debe resguardarse y no ser expuesta bajo ningún motivo. De esta manera, el dueño de las claves puede cifrar cualquier mensaje con su clave privada, compartir el mensaje cifrado y otros actores pueden descifrarlo con la clave pública, asegurándose de que quien cifró el mensaje conserva la clave privada y, por ende, es el dueño de las claves.

La firma digital es un mecanismo que permite comprobar la autenticidad, no repudio e integridad de un mensaje: que el mismo fue generado por la entidad que dice haberlo generado, sin posibilidad de negarlo, y que su contenido no fue modificado luego de ser firmado. Con el uso de claves asimétricas, este mecanismo puede aplicarse fácilmente, siguiendo los pasos del diagrama de la figura 1.4:

i) El emisor toma el mensaje a enviar y calcula su hash con un algoritmo predefinido, obteniendo un valor de largo predefinido.

ii) Luego, encripta ese hash con su clave privada. Debido a que este proceso es costoso computacionalmente, evitamos firmar el mensaje completo, potencialmente mucho más grande que el largo fijo del hash.

iii) Una vez encriptado, se envía el mensaje original junto con el hash encriptado al receptor.

iv) Este receptor, en posesión de la clave pública del emisor, puede entonces descifrar el hash enviado y compararlo con el resultado de volver a calcular el hash del mensaje (utilizando el mismo algoritmo que se utilizó originalmente).

v) Si estos coinciden, el mensaje habrá sido recibido correctamente, sin ser modificado, y estamos seguros de que quien lo envía es quien dice ser.

Este mecanismo permite generar actores en la red, representados por un par de claves pública/privada, que pueden enviar mensajes firmados y los demás participantes pueden verificar que los mismos fueron enviados por este actor en caso de poseer la clave pública asociada.

Cabe destacar que hay numerosos algoritmos para generar estos pares de claves, entre ellos RSA [4] o algoritmos de curva elíptica [5,6], pero los detalles y características de cada uno excede el alcance de este trabajo.

1.2.3. Transacciones en un sistema basado en *Blockchain*

Teniendo claves públicas y privadas, contamos con un mecanismo para que cualquier actor en una red pueda firmar con su clave privada un mensaje, enviarlo a la red y luego ser verificado por los demás actores sin duda alguna de que quien lo envió es quien dice ser.

En una red de criptomonedas, podemos ver a las diferentes billeteras simplemente como un par de estas claves, donde cada vez que quieren realizar una transferencia de una moneda, crean una transacción indicando la cantidad de la moneda a enviar y el receptor, y la firman utilizando su clave privada. Al enviarla a los demás partícipes de la red, si el mensaje es verificado con su clave pública y cuenta con los fondos necesarios, puede ser procesada y efectivizada, asignándole el valor transferido al receptor y quitándoselo al emisor.

Un aspecto clave aquí es cómo los demás actores conocen la clave pública de un emisor, dado que la generación de las mismas suele ser un proceso que sucede por fuera de la red (se conoce esta situación como *off-chain*). Para esto, se puede utilizar la clave pública para identificar al emisor o una derivación de la misma para identificar la cuenta. Así, una cuenta no es más que una clave pública que envía mensajes firmados por su par privado, evitando que haya un registro que asocie cuentas a una clave pública y así hacer más eficiente y escalable la red.

Con este mecanismo, generar una nueva cuenta es un proceso que puede ser realizado de manera *offline*, permitiendo que cualquier entidad genere una nueva sin revelar su identidad. Esta cualidad de anonimato resulta ser uno de los principales valores dentro de la red: aunque todos las interacciones queden registradas públicamente (como transacciones o uso de *smart contracts*), los actores de las mismas son simplemente cadenas de caracteres, por lo que no hay una asociación directa a su dueño. Además de las interacciones, también

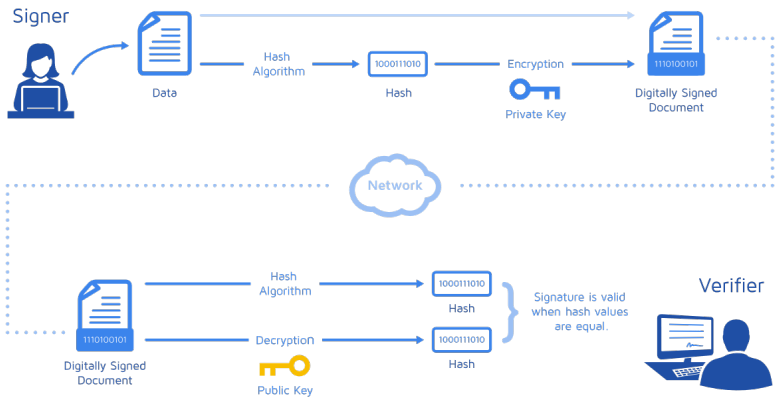


Fig. 1.4: Diagrama del mecanismo de firma digital, donde un emisor firma con su clave privada el hash del mensaje a enviar, transmite ambos valores y los receptores pueden validar con la clave pública del emisor la integridad y autenticidad del mismo, garantizando el no repudio.

es público el contenido de la cuenta, con lo cual alguien con fines maliciosos podría querer encontrar al dueño de una cuenta con gran valor para poder atacarlo.

El hecho de que la creación de cuentas sea libre para los usuarios conlleva a una situación que podría ser considerada como indeseable por la red: cualquiera de ellos puede crear una cantidad ilimitada de cuentas y así realizar acciones maliciosas que abusen de esto (por ejemplo, enviar una gran cantidad de transacciones desde cada cuenta que sobrecarguen a un nodo, resultando más difícil de detectar por venir de muchas cuentas y no de una sola [7]). Para evitar este problema, en algunas redes se pide información adicional del usuario para poder crear una cuenta, lo cual atenta directamente sobre el anonimato de la misma.

1.2.4. *Blockchain* o cadena de bloques

Si bien con lo visto previamente podemos generar transacciones en la red, nos falta un mecanismo para guardarlas de manera que perduren a lo largo del tiempo. Para esto surge la idea de bloques o conjuntos de mensajes de la red (entendiendo que no solo se van a guardar transacciones, sino también otros tipos de valores).

En los sistemas basados en Blockchain, usualmente se utilizan bloques de un tamaño predefinido para ir almacenando las transacciones y, así, mantener un registro público del estado de todas las cuentas. Para mantener un orden, cada nuevo bloque genera una referencia al bloque anterior, formando una cadena de bloques. A su vez, en cada uno de los bloques, se agrega *metadata* sobre el mismo, por ejemplo el número de bloque, fecha y hora, entre otros.

El problema con esta propuesta es que al estar la información distribuida en toda la red, si un conjunto de los actores decidieran modificar la cadena de bloques para, por ejemplo, eliminar una transacción de un bloque, o modificarla para aumentar el valor de la misma, no hay ningún mecanismo que lo impida: siendo suficientes, estos nodos podrían "convencer" a los demás que es cierto lo que sus registros indican.

Luego, la solución para este punto crítico involucra los elementos vistos anteriormente: utilizando una función de hash podemos incluir una firma del bloque anterior de manera que, para verificar que ese bloque no ha sido cambiado, alcanza con computar la función de hash (que recordemos que es *barata* en términos computacionales) del bloque y compararlo con el resultado almacenado en el bloque siguiente. El mecanismo se utiliza de la siguiente manera, representado por la figura 1.5: se genera un primer bloque para la red al que vamos a llamar **bloque génesis**. A partir de éste, el siguiente bloque mantiene una referencia al anterior bloque y además calcula el hash del bloque anterior (usando una función de hash consensuada por el sistema). Esto puede realizarse tomando toda la información del bloque o una cantidad suficiente para que sea un conjunto representativo, siempre incluyendo el hash que este bloque guarda del previo (es decir, el hash del bloque anterior del que estamos calculando el nuevo hash). Este hash se agrega en el bloque actual y se persiste al guardarlo.

Como veíamos en la definición de función de hash, una propiedad sumamente útil es que el valor del hash es siempre el mismo dada la misma entrada, pero si ésta se modifica en lo más mínimo, el resultado del hash cambia rotundamente. Así, si guardamos el hash del bloque anterior y alguien luego decide modificarlo, el hash obtenido del bloque modificado debería cambiar y no coincidiría con el que se ha almacenado en el bloque siguiente. Luego, el atacante no solo tiene que modificar el bloque a atacar, sino que debe recalcular el hash de ese bloque y luego atacar el siguiente bloque para cambiar el hash guardado del bloque

anterior. Pero, como dijimos previamente, el bloque siguiente a éste habrá calculado el hash de este bloque considerando el valor anterior. Por ende, deberíamos también modificar ese valor, y así sucesivamente.

En conclusión, si quisiéramos modificar un bloque, deberíamos modificar todos los bloques de la cadena a partir del bloque a atacar. Entonces, si la complejidad de crear un bloque es tal que no sea algo fácil, un ataque de este estilo resulta impracticable a partir de una cierta cantidad de bloques. Cuanto más *profundo* está el bloque en la cadena, es decir, cuantos más bloques se han agregado posteriormente a un bloque, más difícil y mayor esfuerzo se necesitará para modificarlo.

Este fundamento es la clave para la integridad de una *blockchain*: la información perdura en el tiempo inalterable dada la complejidad computacional para modificarla a partir de una cierta cantidad de bloques. Obviamente, esta complejidad va a estar dada por la manera en la que la red genera los bloques: si son sencillos de generar, podría ser una vía de ataque. Por esto, las redes deben tener la capacidad de adaptar sus protocolos y herramientas para mantenerse actualizados: una propuesta que no es actualizada durante mucho tiempo, puede ser superada por el avance tecnológico y resultar trivial para atacar.

La red va a considerar, entonces, como *fehacientes* aquellas transacciones que han sido guardadas en un bloque y que ya se hayan creado y *apilado* una cantidad de bloques posteriores que las terminen de confirmar (ya que a partir de ese punto debería ser casi imposible que sean modificadas).

Cada una de las redes define la estructura y contenido de sus bloques, pudiendo agregar toda la información que sea necesaria, y definiendo una buena relación entre tiempos de generación y tamaño de bloque: a mayor tamaño, cada bloque contiene mayor cantidad de información, pero tardan más tiempo en generarse (y por ende, tarda más tiempo en confirmar las transacciones).

Finalmente, si bien la idea de una cadena de bloques es el pilar en todas los sistemas basados en *Blockchain*, cada una de ellas implementa de diferente manera cómo decidir qué bloques son los que deben perdurarse en la cadena y cuáles no. Esta manera de decidir en un ambiente descentralizado es llamado **protocolo de consenso** y vamos a ver algunos

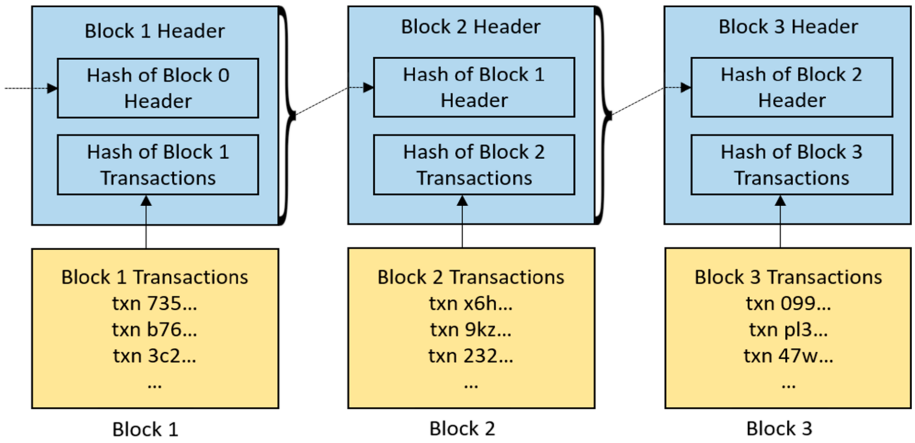


Fig. 1.5: Cadena de bloques donde cada uno almacena el hash de un subconjunto de valores del anterior (excepto el bloque 0 o génesis). La parte celeste se corresponde con la metadata y valores fundamentales del bloque, mientras que la parte amarilla refleja la parte dinámica del mismo: transacciones u otras interacciones que la red permita almacenar allí.

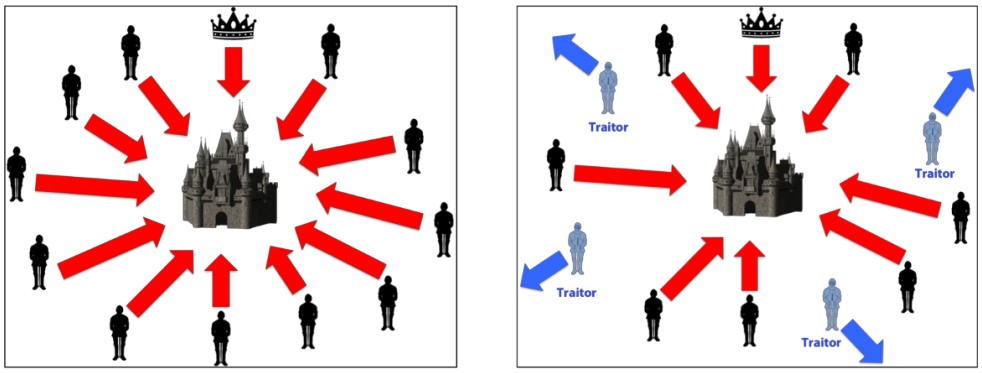
de los más utilizados por las redes más importantes.

1.3. Protocolos de consenso

En una red completamente descentralizada, todos los nodos que participan de ella tienen el mismo nivel de autoridad: no hay nodos más importantes para tomar decisiones (como qué bloque agregar en la cadena), sino que deben ser tomadas en conjunto. Y más importante, esta manera de tomar decisiones debe ser escalable, ser fácil de mantener en el tiempo y soportar ataques de usuarios maliciosos.

Este problema es conocido como el problema de los **generales bizantinos** [8]: un conjunto de generales que asedian una ciudad deben ponerse de acuerdo en si atacar o retirarse a través del envío de mensajes entre ellos. Para comunicarse, cada general envía un mensajero a otro general, creando así un canal de comunicación punto a punto (es decir, no existe la posibilidad de hablar con varios generales a la vez). Para poder ganar, una cierta mayoría de los generales debe atacar, siendo derrotados si la cantidad que ataca es menor a ese valor (por ejemplo, podrían requerir que al menos el 80 % de los generales ataque para ganar).

Encontramos, entonces, dos puntos de falla para el éxito del ataque: por un lado, podría haber un problema en la comunicación entre los generales y que los mensajes no lleguen a sus destinatarios (en términos de red, podría haber un corte en algún enlace y perderse los paquetes). Por otro, suponiendo que los mensajes se envían correctamente, podrían aparecer generales que actúan maliciosamente enviando mensajes contradictorios a los demás. Si todos los generales fueran honestos, situación representada en la figura 1.6(a), todos deciden atacar y el ataque resulta victorioso. Pero, por ejemplo, algunos generales podrían ser traidores y comunicar que quieren atacar, pero finalmente retirarse al momento del ataque. Este escenario, planteado en la figura 1.6(b), provoca que la cantidad de generales que atacan sea menor y el ataque falle (suponiendo que la fuerza coordinada no es suficiente para la victoria).



(a) Generales honestos *(b)* Algunos generales son traidores

Fig. 1.6: Representación del problema de los generales bizantinos. En (a), todos los generales son honestos en su decisión de atacar y ganan el ataque, mientras que en el caso (b) algunos generales son traidores y no atacan, de manera que el ataque falla.

Para ambos escenarios aparecen dos situaciones características. La primera es la de inconsistencia de la información: un general puede enterarse que el general X quiere atacar desde el general Y, pero que indica retirarse desde el general Z. Luego, no tiene manera de

saber qué información es la correcta, ni quién es el general que está actuando de manera maliciosa, llevando a una situación de inconsistencia en la información. Así, un general podría concluir que debe atacar, pero otro que debe retirarse.

El segundo escenario tiene que ver con la toma de decisiones: sea por una actitud maliciosa o por problemas en la comunicación, podría suceder que los mensajes que llegan a un general sean insuficientes para tomar una decisión. Si requieren que el 80 % de los generales decida atacar, pero recibimos mensajes solo del 60 % de los demás generales (suponiendo que votan atacar), no logra llegarse al quorum para definir qué hacer.

Estos dos escenarios generan dos nuevas situaciones donde quienes deciden no atacar puedan provocar la derrota en el caso de que haya un grupo que decida hacerlo. Pero a diferencia de lo visto en el caso de la figura 1.6(b), no todos actúan con malicia, sino que se trata de un problema de comunicación y toma de decisiones en base a la información que recibe cada general. Esta dificultad de coordinar una acción en base a distintos actores que operan independientemente es una propiedad que caracteriza a muchos sistemas distribuidos. Los algoritmos que dan soporte a esta toma de decisiones son conocidos como **protocolos de consenso** y caracterizan a cada una de las redes que los implementan.

1.3.1. *Proof of Work (PoW)*

El protocolo de Prueba de Trabajo (o *Proof of Work*, al que nos referiremos como PoW) es el primer protocolo propuesto para un sistema basado en *Blockchain* en el *whitepaper* de Bitcoin. Este protocolo propone agregarle dificultad artificial a la generación de bloques de manera de que, mediante el trabajo computacional de los nodos, asegurarse de que cualquier acción malintencionada requiera un esfuerzo computacional tan grande que no resulte práctico.

En el caso de Bitcoin, el algoritmo de hash que se utiliza para cada bloque toma como entrada un subconjunto de la información contenida en el bloque y, también, un número entero aleatorio para generar el hash de salida (llamado *nonce*). Considerando esto, el protocolo de consenso se basa en que la red acepta el primer bloque que contenga una cierta de cantidad de ceros en el comienzo del hash del mismo. Por ejemplo, si esta cantidad de ceros debe ser ocho, todos los nodos de la red toman los valores del bloque que corresponden para el hash y van probando con distintos valores de enteros hasta encontrar uno tal que la salida sea un hash que comience con ocho ceros. En el esquema de la figura 1.7 vemos cómo dentro de cada bloque queda entonces guardado también el *nonce* (representado por el campo de asteriscos) que cumple con el requisito de dificultad de la red. Indudablemente, este cálculo es completamente artificial: no agrega valor a la red, sino que solo agrega dificultad para generarlo.

Con este mecanismo (llamado **minado de bloques**), generar un bloque es algo difícil para la red y requiere de una inversión de recursos computacionales para ganar la carrera en la que todos los nodos intentan encontrar un hash que cumpla lo requerido. La dificultad, a su vez, puede ajustarse: si la red detecta que se están generando pocos bloques en un cierto período de tiempo, puede disminuir la cantidad de ceros que se requiere en el hash (y así hacer más fácil el minado). También, si sucede lo contrario, se puede aumentar esta dificultad. La motivación para ser el ganador y proponer un bloque es que quien lo logra recibe una recompensa en moneda.

Una vez que un nodo propone un bloque que cumple con el requerimiento, los demás nodos lo aceptan como válido, se agrega en la cadena, desechando todo el trabajo incurrido por el resto de la red, y se otorga una cantidad fija de moneda de la red a quien propuso

ese bloque (por ejemplo, bitcoins). Se ve, entonces, que para el funcionamiento de una red basada en PoW, existe un derroche intrínseco de recursos computacionales: una gran parte de la red está invirtiendo energía para realizar cómputo que, a lo sumo, puede aprovechar un único minero por bloque. Si bien el minero que logra encontrar el hash que lo habilita a proponer un bloque recibe una recompensa en base a su inversión de recursos computacionales, tanto él como el resto de los mineros del sistema terminan desperdiciando recursos computacionales en cálculo que solo tienen el sentido artificial de darle solidez al sistema. A pesar de este derroche, es una solución muy interesante y original al problema del consenso descentralizado y sentó una base fundamental para el trabajo en el área.

Cabe destacar que existen otras redes con implementaciones propias de *PoW*, como Ethereum 1.0, Bitcoin Cash, Litecoin; donde cada una de ellas decide sus propios métodos para agregar trabajo y los requerimientos de ese trabajo.

Otro problema con este tipo de algoritmos es que los tiempos para generar bloques pueden ser altos: si fuera rápido generar un bloque significaría que es fácil lograrlo y peligraría la integridad de la red. Luego, estas redes suelen tener tiempos por bloque altos (en el caso de Bitcoin es de 10 min) y un usuario debería esperar la aparición de varios bloques para estar seguro que una transacción que se envió a la red no pueda ser revertida.

1.3.2. *Proof of Stake (PoS)*

El aspecto de la dificultad artificial conlleva a buscar otros métodos para la toma de decisiones que no desperdicien recursos y que sean potencialmente más rápidos (evitando tiempos de bloque altos). Para esto, el protocolo de Prueba de Participación (o *Proof of Stake*, al que nos referiremos directamente como *PoS*) acerca una nueva manera de manejar las decisiones: los nodos tienen un peso en la red relativo a la cantidad de moneda que poseen bloqueada en su cuenta. La filosofía detrás de esta decisión de diseño es simple: se supone que quien posee moneda, tiene interés en que el sistema sobreviva y muestre solidez ya que de otro modo perdería valor la moneda que posee. Llevando esta idea aun más allá, se da por sentado que quién posee más moneda, apuesta por la salud del sistema porque involucra su inversión en mayor medida.

Básicamente, en una red de *PoS* participan nodos con cuentas donde se deposita una moneda de la red y, luego, dependiendo de su cantidad de esa moneda en relación a la cantidad total activa de la red, tendrán una mayor probabilidad de tomar decisiones en

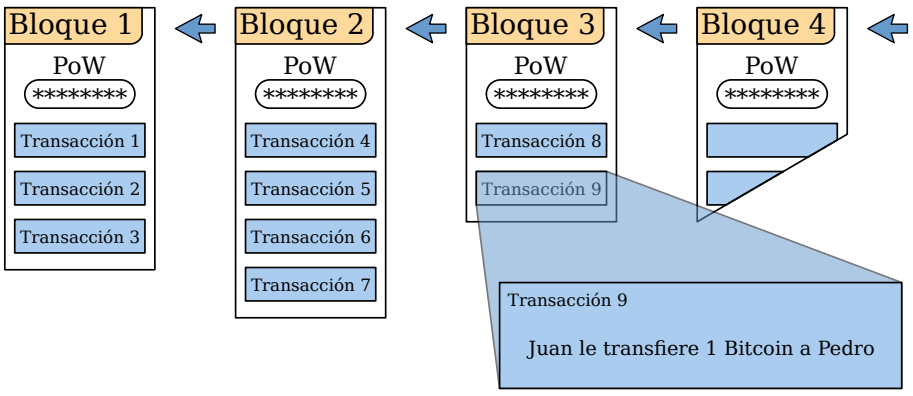


Fig. 1.7: Esquema de blockchain para *PoW*, donde cada bloque contiene el valor utilizado como *nonce* para lograr el hash que cumpla la dificultad de la red.

la misma. Este monto o *stake* asignado a la cuenta del nodo puede estar bloqueado para su uso de manera de asegurar que el nodo participante no pueda quitarlo fácilmente, y hasta podría ser sancionado si su accionar fuera indebido (acción llamada *slashing*). Algunas redes no son tan estrictas y permiten que los nodos hagan uso de esas monedas sin bloquearlas. Nuevamente, a quien cree el siguiente bloque se le otorga una recompensa para justificar la participación en la red y hacerlo económicamente provechoso. Si bien de esta manera evitamos el trabajo innecesario, ahora tenemos un nuevo problema: cómo elegir quién es el que toma las decisiones en cada ronda de la red. Una idea podría ser que la red no se encuentre totalmente descentralizada, sino que hayan nodos designados para hacer la elección de quién es el que propone el siguiente bloque en base a sus *stakes* utilizando algún algoritmo de selección aleatoria que considere ese peso. Como mejora a esta idea, esos nodos designados podrían no ser fijos, sino que serían seleccionados en cada ronda bajo algún criterio (aleatoriamente, por *round robin*, u otros) de manera que se evite centralizar el poder dentro de la red.

La idea de que la participación de cada nodo sea proporcional a la cantidad de *stake* que mantiene en la red conlleva un nuevo problema: los nodos con mayor cantidad de *stake* son los que van a tomar la mayor cantidad de decisiones (motivo por el cual hay autores que lo consideran un mecanismo defectuoso, al ser plutocrático, oligopolístico y permisionado [9]). Así, se produce una centralización de la red que va en contra del fundamento de descentralización de estos sistemas, algo que no sucedía con los algoritmos de *PoW* al no depender de nada más que el poder de cómputo de los nodos (algo debatible si consideramos los *pooles* de minería). Pero lo positivo es que las decisiones son sumamente más rápidas, logrando tiempos por bloque de hasta milisegundos.

Entre los primeros en implementar *PoS* en su red está `PeerCoin`, pero luego fueron surgiendo muchas otras redes que con diferentes variaciones, entre ellas `NXT`, `Tron`, `Solana` y `Algorand`. Notablemente, `Ethereum` también migró a un protocolo de *PoS* durante el desarrollo de esta tesis, dejando de lado el *PoW* y así disminuyendo drásticamente el consumo energético de su red.

Para el desarrollo de esta tesis nos interesa experimentar con una red de *PoS* de código abierto que nos permita implementar instancias privadas de manera de tener total control de la red, sin depender de redes experimentales públicas. Decidimos trabajar con `Algorand` dado que es una solución de código abierto con muy buena documentación, un gran enfoque académico, una comunidad de desarrolladores muy activa, y proponen una solución interesante para solucionar la distribución de las decisiones.

Otras de las redes consideradas para este análisis fueron `NXT`, `Tron` y `Solana`, pero fueron descartadas por falta de documentación, código incompleto o comunidades muy poco activas. También se consideró el uso de `Ethereum 2.0`, pero el desarrollo de la misma se encontraba en proceso y no contábamos con una buena base para poder realizar experimentos.

1.4. Algorand

Fundada en 2017 por el profesor del MIT Silvio Micali, `Algorand` [10] es una red basada en Blockchain con un protocolo de consenso basado en *PoS* llamado *Pure Proof of Stake*. Se trata de un proyecto con código abierto que es mantenido por la Fundación `Algorand` y cuya red principal fue lanzada en Junio de 2019. El gran motor innovador del proyecto se basa en utilizar herramientas criptográficas para resolver los problemas de consenso de

la red, "escalando" el algoritmo de acuerdo bizantino. La moneda nativa para el manejo de la red es llamada `algo`.

1.4.1. Innovación

La red se propone solucionar sin compromisos lo que llaman el *"blockchain trilemma"*: seguridad, escalabilidad y descentralización [11].

El corazón de `Algorand` está en la función aleatoria verificable (*Verifiable Random Function*, o VRF [12]): una función que dada una entrada (incluyendo un mensaje y valores secretos de quien lo crea) genera una salida que es un mensaje firmado por la cuenta con un hash numérico. Este hash es un número que, cuanto más pequeño sea, más chances tendrá de ser elegido en las decisiones de la red.

Utilizando esta función, los nodos pueden generar de manera *offline* pruebas de los mensajes que envían a la red, y son estos mismos mensajes los que serán tenidos en cuenta por los demás nodos para elegir quiénes van a proponer bloques en cada ronda.

Uno de los puntos claves de esta función es la capacidad de producir una prueba de manera *offline*: no se necesita comunicarse con la red para poder generar la participación del nodo en una ronda, sino que puede realizarlo por fuera de la misma y luego enviar el mensaje cifrado a la red. Este mensaje cifrado es justamente verificable por los demás y permite que en el protocolo de consenso se pueda ahorrar mucho tiempo en comunicación ya que cada nodo va a generar su participación ya firmada y verificable.

Esta implementación criptográfica resulta ser el avance más interesante que plantea y justifica la creación de `Algorand`. La función *VRF* toma como parámetros una clave secreta y un valor (como la clave de participación secreta de un nodo y la *seed* de selección, valor que se genera al crear la cuenta del nodo) y retorna un valor pseudoaleatorio en conjunto con una prueba que cualquiera puede utilizar para verificar el resultado.

La función es la misma en todos los nodos de la red, con lo cual cada uno de ellos puede comprobar lo obtenido por otro de los nodos de manera independiente sin necesitar más información que la recibida en el mensaje.

Otro punto importante de la *VRF* utilizada en `Algorand` es que es de código abierto [13] y, si bien su implementación no es parte del alcance de este trabajo, resulta igualmente de gran interés para un análisis criptográfico.

1.4.2. *Pure Proof of Stake (PPoS)*

`Algorand` plantea un algoritmo de consenso que se llama *Pure Proof of Stake (PPoS)* que intenta resolver el problema de que los nodos con mayor cantidad de *stake* controlen la red, escalando así el acuerdo bizantino [14,15]. Como veíamos previamente, el algoritmo se basa en la implementación de una *VRF* que permite calcular, para un cierto evento (cuyo detalle veremos luego), si ese nodo debe o no participar del mismo de manera *offline*. Estos nodos elegidos forman un comité con la finalidad de decidir sobre ese evento.

La probabilidad de participar es la misma para cualquier unidad de `algo` de cada nodo, pero cada actor de la red la ejecuta tantas veces como cantidad de `algo` tiene. Así, si una cuenta contiene solo 1 `algo`, ejecuta una sola vez la función, quedándose con el resultado de la misma. Pero si contiene, por ejemplo, 10 `algo`, ejecuta diez veces la función, incrementando sus posibilidades de ser elegido. En otras palabras, cada *algo* de la cuenta se corresponde con un ticket de lotería para participar del comité. Las cuentas con mayor

stake van a ser las que más cantidad de veces sean elegidas para conformar el comité, pero también le asegura tener posibilidades a las cuentas con menor cantidad de `algo`.

Con esta estrategia, `Algorand` divide el protocolo de consenso en distintas etapas, de manera de darle acceso a las cuentas con menor cantidad de *stake* a participar del mismo: durante la escritura de esta tesis hemos visto en la red principal la participación en el consenso de cuentas con 0,1 `algo`, monto que actualmente se encuentra por debajo de los diez centavos de dólar.

Para agregar aleatoriedad al proceso de ejecución de la *VRF*, cada bloque nuevo genera una nueva *seed* que es utilizada como parámetro en la siguiente ejecución de esta función. Recién con la aparición del bloque se conoce este valor, de tal manera que los nodos no pueden precalcular su valor y así se agrega un factor aleatorio en los cálculos.

Veamos entonces las distintas etapas del protocolo de consenso.

Etapa uno: propuesta de bloque En esta primer etapa, ilustrada por la figura 1.8, se debe decidir quiénes propondrán un nuevo bloque y cuál será el mismo. Para esto, todos los nodos que participan del consenso ejecutan la *VRF* tantas veces como corresponda según su cantidad de `algo`, determinando con su resultado si la cuenta fue seleccionada. En el caso de ser elegida, el nodo envía a la red el bloque propuesto junto con la salida de la *VRF*, la cual prueba que la cuenta fue elegida para proponerlo.

El resultado de esta etapa es que cada cuenta elegida envía un bloque a la red para ser tratado.

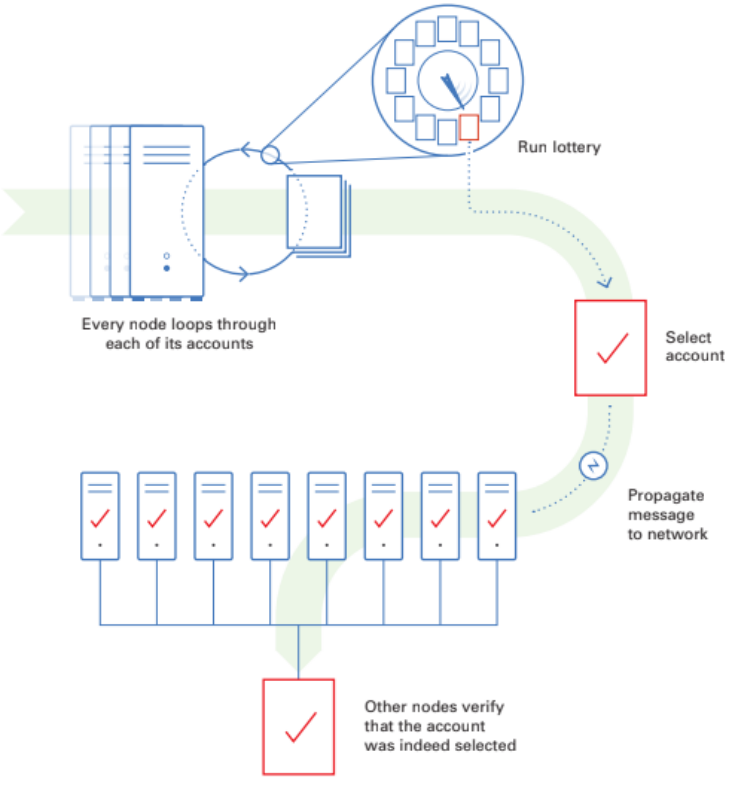


Fig. 1.8: Etapa uno del protocolo de consenso: las cuentas de la red ejecutan la *VRF* y aquellos que son seleccionados proponen un nuevo bloque al sistema. Reproducido de la documentación oficial de `Algorand` para desarrolladores [16].

Etapa dos: voto suave (o soft vote) Contando con todos los bloques propuestos por estas cuentas, ahora corresponde elegir cuál de ellos se va a agregar efectivamente a la cadena, sólo uno podrá avanzar en el proceso. A medida que los nodos reciben estos bloques, los verifican utilizando la *VRF* y, como mecanismo de elección, se quedan siempre con el que tenga el valor de hash más pequeño (interpretando el hash como un valor numérico). A su vez, solo es retransmitido el bloque recibido si este valor es menor que los recibidos previamente, lo que disminuye el tráfico de red.

Luego de un tiempo predeterminado, la red termina de propagar el bloque con el menor hash de los propuestos y ese es el elegido para continuar el proceso. La figura 1.9 hace referencia a este proceso.

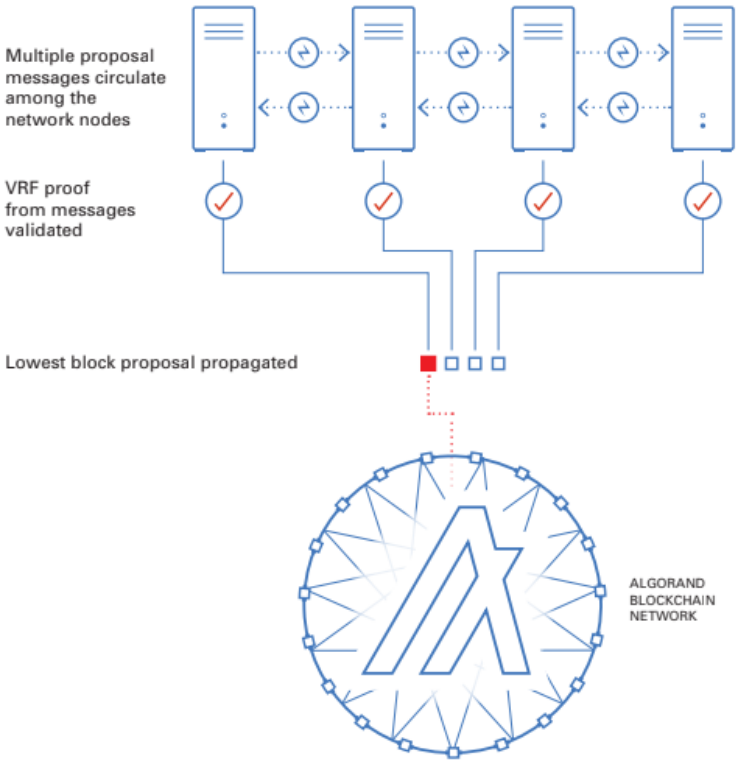


Fig. 1.9: Etapa dos del protocolo de consenso: las cuentas propagan el bloque con el menor hash de *VRF*. Reproducido de la documentación oficial de `Algorand` para desarrolladores [16].

Nuevamente, todos los nodos ejecutan la *VRF* para cada cuenta y así formar un comité para emitir votos con el peso de su *stake* sobre el bloque propuesto, siguiendo lo mostrado en la figura 1.10. Este tipo de votos los llamamos votos *suaves* (o *soft-votes*), donde la decisión se toma considerando el peso de cada votante (es decir, su *stake*), en lugar de ser simplemente un recuento de mayoría. Luego, las cuentas con mayor *stake* influyen en el resultado de manera más fuerte que aquellas con menor *stake*. La idea de este mecanismo es que quienes tienen mayor *stake* tienen más para perder si actuaran maliciosamente, habiendo invertido en mayor medida en la red.

Las cuentas elegidas para el comité generan sus votos y estos se envían al sistema junto con la salida de la *VRF* para poder verificar su participación.

Una vez que se alcanza un cierto quorum de votos, la red ya se encuentra en condiciones de pasar a la siguiente etapa.

Etapa tres: certificación de voto Ya teniendo el bloque propuesto que ha sido seleccionado y aprobado por un cierto quorum, el último paso es crear un nuevo comité que verifique el bloque para detectar acciones maliciosas (como cuentas transfiriendo más monedas de las que poseen o haciendo ataques de *double spending*, entre otras). Esto se realiza, al igual que en los casos anteriores, ejecutando la *VRF* en el nodo para todas sus cuentas, situación ilustrada por la figura 1.11.

Las cuentas elegidas votan sobre el bloque, se envían los votos a la red junto con la prueba de la *VRF* y al llegar al quorum, el bloque se certifica, siendo agregado a la cadena de `Algorand`. Una vez agregado, el proceso comienza nuevamente.

Consideraciones generales del protocolo El tamaño de los comités y el quorum necesario varía para cada paso, siendo un valor que puede configurarse en los clientes de la red en caso de ser necesario. Además, en cada paso se conforma un nuevo comité, de manera que no son siempre las mismas cuentas las que toman las decisiones.

Para que este protocolo funcione, `Algorand` calcula que más de dos tercios de los participantes ponderados de la red deben ser "honestos" (es decir, considerando su *stake*). En su implementación, se eligió un valor del 80 % para este indicador, determinando así el tamaño de sus comités y cantidad de votos: si se eligen pocos integrantes del comité, sus votos tienen mayor peso y un atacante podría realizar acciones maliciosas más fácilmente, pero si se agranda demasiado el comité se incrementa la dificultad para conformarlo y

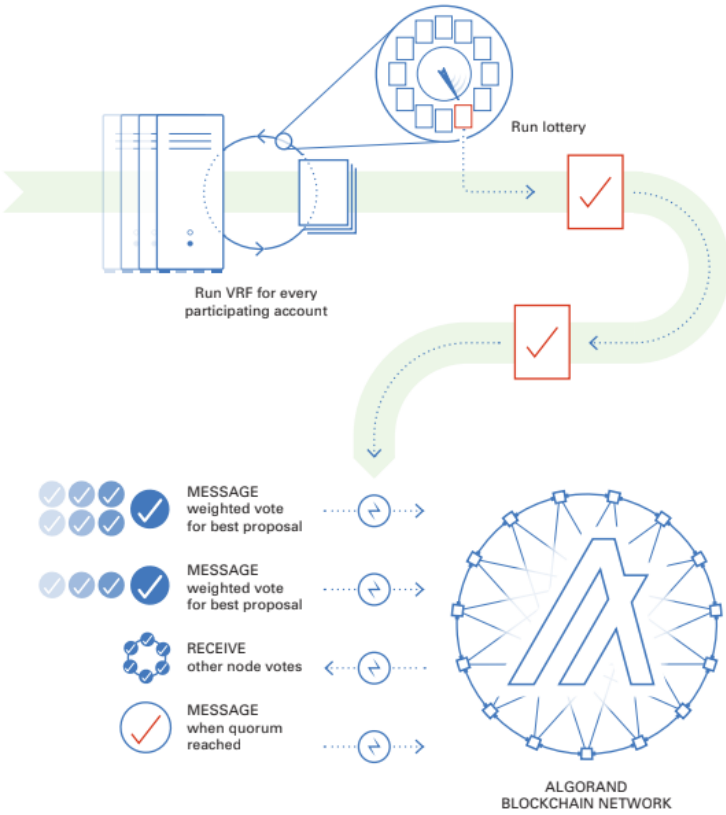


Fig. 1.10: Etapa dos del protocolo de consenso: las cuentas corren la *VRF* y aquellos seleccionados votan sobre el bloque elegido para esa ronda. Reproducido de la documentación oficial de `Algorand` para desarrolladores [16].

tomar decisiones en tiempos bajos.

Si `Algorand` no puede conformar los comités porque la cantidad de participantes es insuficiente (por ejemplo, si un nodo con un *stake* alto está fallando) o si no se llega al quorum en algún paso, la red entra en modo de recuperación después de un tiempo predeterminado. Durante ese tiempo, la red no progresa y no se crean nuevos bloques, en cambio los nodos intercambian mensajes de recuperación para intentar retomar el avance. Por ejemplo, si hay más de un 20 % del *stake* de la red que no se comporta debidamente, la red no garantiza su progreso.

Otro elemento a tener en cuenta es que `Algorand` no implementa *slashing*, es decir, no castiga a los nodos que se comporten maliciosamente o que no cumplan con lo requerido en cada ronda. En otras redes, este mecanismo provee una manera de aplicar una sanción a los nodos que se comporten erróneamente, pudiendo perder parte o la totalidad de su *stake*.

1.4.3. Arquitectura de la red

La red de `Algorand` divide sus nodos en dos tipos: nodos participantes de la red y nodos *relay*. A los primeros nos referiremos como nodos y a los segundos como *relays*. La idea es que los *relays* son los encargados de recibir y reenviar todos los mensajes que reciben de los nodos y de sus *relays* conectados, haciendo justamente el *relay* de los mensajes. Este subconjunto de nodos están curados por la Fundación `Algorand` y tienen requisitos de *hardware* específicos dado que son el esqueleto de la red. A su vez, el listado de *relays* es público y cualquiera puede encontrar sus IPs.

Los *relays* se conectan solo entre ellos, eligiendo otros cuatro *relays* del sistema (de este listado público). Durante esta conexión, intentan siempre mantenerse conectados a los mejores *relays* disponibles, midiendo el desempeño de los que tiene actualmente conectados y así analizando si le conviene cambiar a otros.

Los otro nodos, que pueden participar del consenso o simplemente participar pasivamente de la red, eligen automáticamente cuatro *relays* del sistema e intercambia mensajes

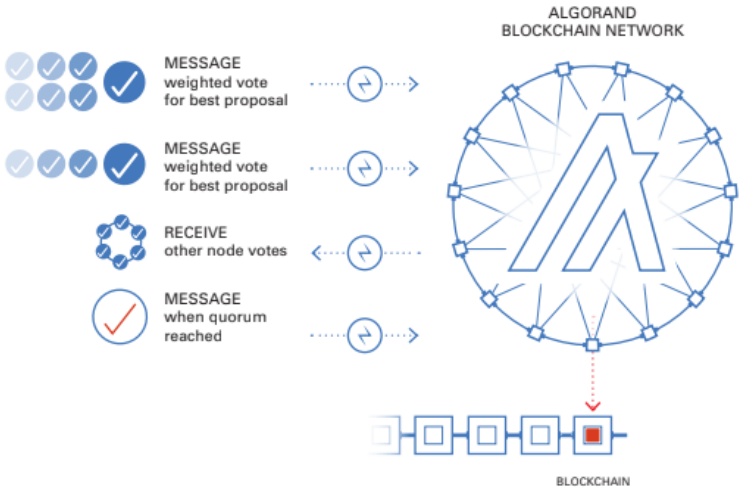


Fig. 1.11: Etapa tres del protocolo de consenso: se crea un nuevo comité que verifica el nuevo bloque y votan sobre el mismo para ser agregado a la cadena. Reproducido de la documentación oficial de `Algorand` para desarrolladores [16].

con ellos. Solo se conecta a estos *relays* y no a otros nodos. Entonces, cualquier actualización de la red la recibirán de sus *relays* conectados. Los nodos también miden a sus *relays* conectados, pudiendo cambiarlos si detectaran problemas con los que tiene en ese momento.

Ambos tipos de nodos siempre se encontrarán activamente conectados a cuatro *relays*, siendo algo parametrizado en el cliente de `Algorand`, buscando permanentemente mantener esa cantidad durante su ejecución (por ejemplo, al comienzo de la red o si alguno de los *relays* conectados se desconecta, el cliente automáticamente elige nuevos para conectarse hasta llegar a cuatro conexiones activas).

Luego, la distribución de la red consiste en nodos conectados a *relays* y *relays* conectados a otros *relays*. Por supuesto, estas conexiones son puramente lógicas, quedando abstraída de la topología física real.

Los nodos se conectan a cuatro *relays* cada uno (pudiendo coincidir en algunos de ellos), mientras que los *relays* se conectan con otros cuatro *relays*. Si bien el *relay* que inicia la conexión va a elegir hasta cuatro otros *relays*, el *relay* que recibe la conexión también recibe información por ese enlace.

Algo importante sobre el mecanismo de envío de mensajes en los *relays* es que realizan *flooding* hacia todos sus pares conectados, pero no reenvían mensajes a quién se los envió, ni si detectan que se trata de un mensaje viejo o inválido. Esta comunicación se implementa en la capa lógica de los nodos y construye un protocolo de *gossip* entre ellos.

1.4.4. Estado actual de la red

La red de `Algorand` mantiene alrededor de 1200 nodos activos a la fecha, de los cuales solo un subconjunto de ellos participa activamente del consenso: alrededor de 170 cuentas participan en las votaciones, de manera que alrededor de un 15 % del total de los nodos interviene en las votaciones.

Estos valores se modifican constantemente, al comienzo de esta tesis era mayor la cantidad de cuentas que participaban, cercano a las 300, pero posiblemente debido a la caída de los mercados de criptomonedas, este valor descendió a casi la mitad.

Con respecto a los *relays*, actualmente `Algorand` cuenta con 110 corriendo en la red, cuyo registro es público y puede listarse corriendo el comando:

```
nslookup -q=srv _algobootstrap._tcp.mainnet.algorand.network
```

A partir de esta información, podemos acceder a las IPs donde están corriendo dichos *relays*.

Algo importante a tener en cuenta con respecto a la cantidad de nodos es que cada uno de ellos puede correr todas las cuentas que quiera, pero cada cuenta puede estar asociada a un solo nodo. Además, hay cuentas que participan indirectamente del consenso delegando su *stake* en otras. Por ello, algunas cuentas pueden contar con mucho *stake*, siendo una parte propio, y otra parte delegado por otras cuentas. Finalmente, si un nodo corre varias de estas cuentas, va a estar manejando una proporción de *stake* muy importante.

Tiempo de ronda

Considerando los datos brindados por `Algorand`, podemos ver que el tiempo de ronda promedio se encuentra alrededor de los 3,3 s para la versión actual del cliente. Para la versión utilizada en esta tesis, según documentación y datos provistos por la comunidad,

el tiempo promedio se encontraba en 4,5 s. Sin embargo, tomando una muestra de 10 000 bloques para ese momento, pudimos calcular el tiempo real en 4,36 s con una muy baja cantidad de transacciones por bloque (130 en promedio, entendiendo que en cada uno podrían aparecer hasta 5000). Considerando el tiempo actual, la disminución en el tiempo por ronda demuestra la capacidad para mejorar que tiene la red.

En nuestros experimentos, va a resultar necesario poder determinar si el sistema se encuentra funcionando dentro de los parámetros esperados. Dada la naturaleza aleatoria de muchos de los eventos que ocurren en el normal funcionamiento, se necesita encontrar una magnitud que nos permita evaluar si el comportamiento de la red de los experimentos se comporta de manera similar a la red real.

Como queremos saber si la red progresa, creando nuevos bloques, podemos considerar el tiempo por ronda como un buen indicador. Con lo cual, si nuestra red experimental tiene un tiempo de ronda cercano a los 4,36 s con bloques vacíos o con pocas transacciones, sabremos que el ritmo que lleva se condice con lo esperado de una red real. En cambio, si notamos tiempos de ronda más altos, será un indicador de que podría haber algún problema en la ejecución del sistema, sea por malas configuraciones, nodos funcionando incorrectamente o simplemente porque el *hardware* no es suficiente para la cantidad de nodos a emular.

Para ver el valor del tiempo por ronda mientras se ejecuta el experimento, contamos con una aplicación provista por `Algorand` llamada `carpenter` que nos permite ver los *logs* a medida que se generan con facilidades para su lectura. Luego, contamos con herramientas propias para tomar estos *logs* y analizar los tiempos durante toda la duración del experimento.

Distribución del *stake*

La distribución actual del *stake* en `Algorand` es un dato que podemos extraer de repositorios públicos de información donde se listan las cuentas por monto en orden decreciente [17]. La información que requerimos para esto es el listado de cuentas participantes del consenso y el *stake* de cada una. Esta información nos permite tener una foto de cómo está distribuido el *stake* en la red actual.

Al tiempo de elaboración de este documento, la red de `Algorand` cuenta con alrededor de 170 cuentas participantes del consenso. Según los datos extraídos con los que confeccionamos la figura 1.12, el 80 % del *stake* que participa del consenso, que alcanza para tomar las decisiones de la red, se encuentra concentrado en tan solo 36 cuentas. A su vez, casi un 22 % del *stake* lo manejan seis cuentas, con lo cual si un atacante las afectara alcanzaría para detener el avance de la red. El listado completo puede observarse en la sección A.1 del apéndice.

No estamos incluyendo muchas cuentas que participan del consenso con una cantidad de *stake* muy baja, con lo cual muy esporádicamente producen un bloque. Estos nodos no son de interés para nuestro estudio dado que la cantidad de mensajes y decisiones que aportan a la red es despreciable.

En cada experimento nos va a interesar la relación entre la cantidad de bloques propuestos por nodo y el *stake* de cada uno de ellos. Según lo visto, si todos los nodos de la red mantienen el mismo *stake*, se esperaría que todos los nodos tengan la misma participación en la misma y la cantidad de bloques propuestos por nodo sea equivalente (dada la duración del experimento, no se espera que sean todas iguales por motivos de aleatoridad, pero que sí sea lo más parecida posible). En cambio, en un experimento donde el *stake*

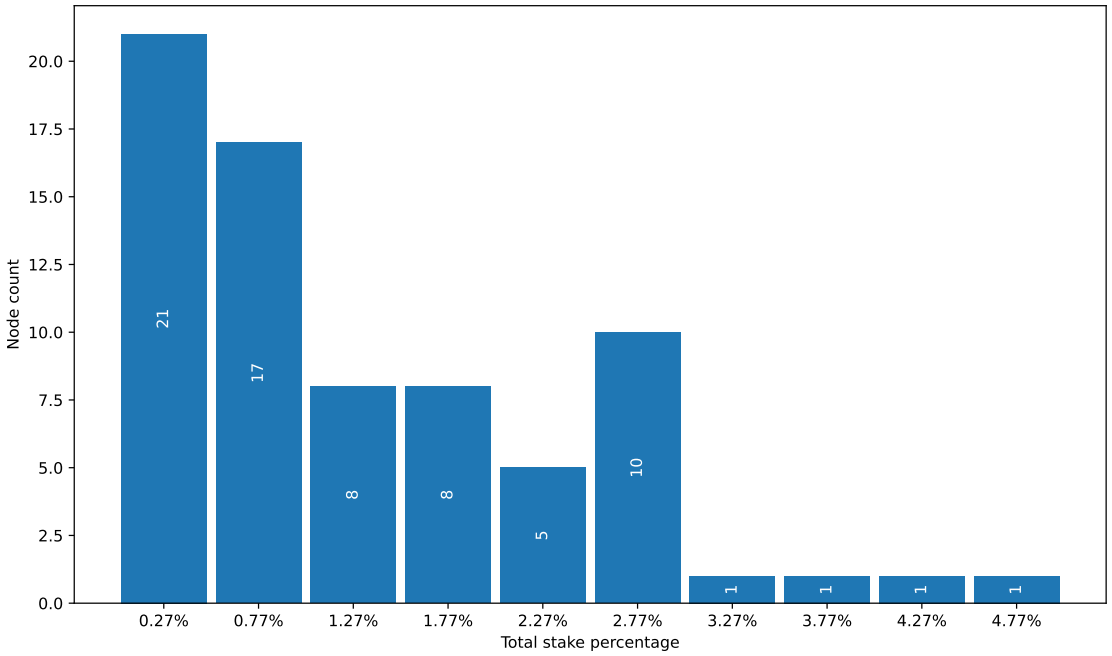


Fig. 1.12: Distribución del *stake* en la red actual de `Algorand`, según porcentaje de *stake* activo en el consenso, en base a datos del 6 de noviembre de 2022. Se marca el valor medio de cada intervalo, con un ancho de *bin* del 0,45 %.

esté distribuido heterogéneamente, la cantidad de bloques propuestos por nodo debe ser proporcional al *stake* del mismo.

1.4.5. Nodos en `Algorand`

Para correr un nodo en `Algorand` necesitamos, además de los binarios del cliente, un conjunto de configuraciones que van a determinar el funcionamiento del mismo. Los binarios se crean a partir de compilar el código del cliente de `Algorand`, mientras que la configuración dependerá de la instancia que quiere ejecutarse: sea para la red principal, una de las redes de prueba o una instancia privada.

Estas configuraciones se establecen a partir de diferentes archivos, y con la ejecución del cliente también se crean nuevos donde se consolida información y *logs*. Algunos de los archivos principales de esta estructura son (en orden alfabético, no de importancia):

- `algod.token`: contiene el token de administración para el nodo. Cualquier acción que quiera realizarse desde fuera del mismo, como enviarle una transacción por vía *REST*, requiere que se incluya este token como medio de autenticación.
- `algod-out.log`: archivo de *log* del cliente. Contiene los mensajes de error y éxito en la ejecución del cliente, pero no contiene los *logs* operativos del nodo.
- `config.json`: incluye la configuración principal del nodo, entre ellos la versión del mismo, límites de conexiones vivas, puertos a utilizar, niveles de *logs*, o si el nodo debe operar como *relay*. A partir de este archivo pueden elegirse una gran cantidad de opciones para la ejecución del nodo, disponibles en su documentación [16].

- `genesis.json`: es el archivo con el génesis de la red, donde se listan todas las cuentas con sus *stakes* iniciales, además de otras configuraciones como la versión de protocolo de consenso a utilizar. Algo importante de este archivo es que también indica qué cuentas están participando del consenso inicialmente y con qué claves públicas (además de definir la duración de las mismas), de manera que al comenzar la red, éstas serán las que toman las decisiones desde el inicio. Para nuestros experimentos ya configuramos todas las cuentas de nuestra red en este archivo de manera que no haga falta registrarlas, ya se encuentran habilitadas desde un principio.

- `kmd-v0.5/`: carpeta que contiene la configuración y el *token* para el manejo del cliente `kmd`, encargado de la parte criptográfica del nodo. Este cliente puede ejecutarse por fuera del nodo de manera que todo lo que sea firmado de mensajes con la clave privada de la cuenta suceda en otro ambiente que no coincida con el público del nodo. Aquí también se encuentran las billeteras o cuentas creadas para el nodo.

- `node.log`: *log* del nodo, con detalle de bloques propuestos, etapas de las rondas y otras interacciones de la red, junto con marcas temporales. El nivel de este *log* puede ajustarse en la configuración, de manera de poder ver en tiempo real un gran nivel de detalle, pero con el costo de un crecimiento mucho mayor de este archivo.

- Carpeta de red (en nuestro caso `privatenet-v1`): es la carpeta que se corresponde con la instancia de la red donde va a correr el cliente. Contiene el listado de cuentas que administra el nodo y las claves asociadas a cada una, entre otros valores.

- `phonebook.json`: si bien este archivo se ubica directamente a nivel de los binarios del cliente (y no en la carpeta del nodo en sí), permite especificar qué otros nodos o *relays* del sistema conoce el cliente. Así, podemos configurar que para un cierto nodo, sólo conozca un listado concreto de *relays* y evitar que se comunique con todos los disponibles.

Además de estos archivos, encontramos otros *logs* y *caches* secundarios que no resultan de interés para resaltar, pero que hacen al correcto funcionamiento del cliente.

Registro de una cuenta en la red

Si bien en nuestros experimentos vamos a definir cuáles son las cuentas participantes de la red ya desde el génesis de la misma, podríamos crear y correr nuevas durante la ejecución de la red. Para lograr esto, dada una nueva cuenta que quiere participar del consenso, debe generar una clave de participación y luego una transacción firmada donde registre esta clave como *online* (es decir, que participa del consenso). Este proceso puede realizarse de manera *offline*.

Luego, elige un nodo para correr esa cuenta, le envía la clave de participación y la transacción, y el nodo se encarga de registrarla internamente y compartir la transacción con la red. Al compartir la transacción, los demás nodos conocen al nuevo participante y, luego de 320 rondas, se aplica el cambio. Por ejemplo, si la transacción es confirmada en la ronda 1000, recién en la ronda 1320 la cuenta podrá participar del consenso.

Con este cambio, el *stake* de la nueva cuenta pasa a encontrarse *online* y se lo considera para los cálculos del quorum necesario para las decisiones.

La clave de participación que genera la cuenta tienen una cierta duración definida en el momento de su creación, permitiendo que una cuenta las renueve periódicamente de

manera de fortalecer la seguridad de sus participantes. El hecho de que estas claves y transacciones se puedan generar además de manera *offline* agrega a la seguridad de estas cuentas, no debiendo exponer datos altamente críticos como sus claves privadas.

Así como puede registrarse en el consenso, la cuenta puede darse de baja del mismo para no participar más, removiendo así su *stake* de la red.

El proceso para esto es similar al anterior, la cuenta esta vez firma una transacción para registrar como *offline* su participación, lo anuncia al nodo que corre su cuenta y éste lo envía al resto de la red. Al igual que el caso de registro, este cambio es aplicado luego de 320 rondas, debiendo participar en la red hasta que se cumpla ese período (antes de eso la cuenta permanece activa).

Como el *stake* de la cuenta que se registra como *offline* deja de participar del consenso, la red continúa su normal funcionamiento sin ser afectada: ahora se necesitará una menor cantidad de *stake* para lograr el consenso, en vistas de que se necesita un porcentaje de la cantidad que se encuentra *online*.

Particiones y modo de recuperación

Como fue descrito previamente, en el caso de que la red no logre el quorum para completar alguno de los pasos del protocolo de consenso, los nodos pasan a modo *recovery*, pausando el progreso de la red hasta recuperar el funcionamiento correcto de la misma.

Así, evita generar *forks* de la cadena principal, dado que si la red se parte y no alcanza los valores requeridos, no avanza hasta recuperarlos. Lo negativo de este aspecto es que entonces la red se detiene completamente hasta que se solucione el inconveniente que la detuvo, pero apuestan a que si un atacante estuviera detrás de esto, debería ser muy costoso para el mismo mantenerlo por mucho tiempo.

El impacto de un ataque de ese estilo, que particiona la red, no solo depende del tiempo de interrupción del progreso durante el mismo, sino también del tiempo de recuperación de la red. Si un ataque de pocos minutos logra que la red luego deba estar horas recuperándose, resulta económicamente práctico hacerlo. Pero si la recuperación toma pocos segundos, solo importa el tiempo que transcurre durante el ataque.

Con esta estrategia, `Algorand` apuesta a tener tiempos de recuperación muy bajos que fuercen a que el atacante tenga que mantener activa su acción para pausar la red. Durante ese tiempo la red no avanza, pero apenas se levanta el ataque, retoma su accionar velozmente. Así, se aseguran que no se generan bloques con información maliciosa y los fondos de las cuentas se encuentran seguros, independientemente del tiempo de duración del ataque (sea minutos, horas o días), resultando en una acción cara para el atacante.

`Algorand` logra tiempos de recuperación de segundos implementando su protocolo de recuperación que se dispara cuando los nodos detectan que no se están generando bloques luego de un cierto período de tiempo, sea porque tienen un problema de conectividad a la red, porque no se alcanza el quorum por otro nodo afectado o porque la red se particionó. En este modo, los nodos envían mensajes de recuperación y son propagados por toda la red. Al acumular una cantidad dada de ellos, los nodos se sincronizan y pueden continuar produciendo bloques.

Cuando un nodo se desconecta de la red, pero ésta continúa su progreso, al reconectarse consulta a sus pares por el bloque actual. En el caso de ser posterior al último bloque guardado por el nodo, debe recuperar la información sobre el progreso de la red durante el período que estuvo desconectado, verificar todos esos bloques y finalmente puede volver a participar de la red.

A diferencia de esto, si la red se particiona de manera tal que se detiene la generación de bloques, todos los nodos pasan a modo recuperación, debiendo lograr un quorum dado para poder reanudar el avance. Esto resulta entonces más complejo que el caso de un solo nodo desconectándose de la red por algún motivo (sea por una falla técnica del mismo o producto de un ataque puntual sobre el mismo). En ambos casos, si el nodo atacado tiene un *stake* alto, se puede detener la red durante el tiempo que dura el ataque, pero apenas se agota el mismo, la red se recupera rápidamente.

Más allá de las medidas de desempeño que toman los nodos y los tiempos por bloque, la red no implementa otros protocolos de control sobre el funcionamiento de los nodos y *relays*. Por ejemplo, si un *relay* se encuentra afectado y comienza a enviar mensajes *basura*, la red no contempla un mecanismo específico para descubrirlo y sancionarlo (o al menos notificarlo a otros nodos).

1.5. Trabajo relacionado

En este trabajo de tesis el objetivo será experimentar con la red de `Algorand`, primero realizando una investigación sobre antecedentes de estudios en redes de *blockchain* que nos permitan adentrarnos en el tema.

Para comenzar nuestro análisis consideramos primero trabajos previos con redes de *Proof of Work* de manera de entender sus enfoques y metodologías al ser el mecanismo de consenso más importante del ecosistema, y, por ende, el más estudiado. Luego, realizamos lo mismo para trabajos con redes de *Proof of Stake*, pudiendo diferenciar las estrategias y análisis de ellos con las del protocolo anterior. Finalmente, buscamos investigaciones existentes sobre `Algorand` para encontrar qué aspectos de la misma fueron estudiados y cuáles no.

En trabajos previos hemos visto análisis de la red de Bitcoin, tomando distintas cualidades de la misma para revisar comportamientos y proponer mejoras. Entre estos temas se encuentran analizar el tiempo de propagación de bloques y generación de *forks* en la red (Decker, Wattenhofer [18]), el impacto de la topología física en estos tiempos de propagación (Donet et al. [19]). En ambos trabajos se encuentra una correlación entre las variables estudiadas, pudiendo encontrar mejoras para las mismas.

Otro trabajo que nos resulta de interés es el de Miller et al. [20], donde se analiza la distribución de los nodos en la red. A partir de un cliente de Bitcoin modificado que envía mensajes específicos a otros nodos, logran detectar los nodos más relevantes de la red y así intentar reconstruir la topología lógica de la misma. Entre los valores a descubrir se encuentran la cantidad de conexiones por nodo y la ubicación de *pools* de minería.

Luego, encontramos otros trabajos sobre la escalabilidad en Bitcoin, considerando sus protocolos de red, consenso y otros (Croman et al. [21]), o el *throughput* de transacciones (Sompolinsky y Zohar [22]). Así también, encontramos estudios teóricos sobre la relación entre tiempo de generación de bloques y tiempo de propagación (Garay et al. [23]), y el impacto en seguridad al tener tiempos de bloque muy cortos (Kiayias et al. [24, 25]).

Marco Vanotti [26] analiza los efectos de disminuir el tiempo entre bloques y aumentar el diámetro de la red respecto a la generación de *forks* en la red. Utiliza `Mininet` y `Maxinet` para emular las topologías de red descriptas en los trabajos de Donet et al. [19] y Miller et al. [20].

Silvio Vileriño [27] diseña una metodología para el estudio de sistemas basados en

blockchain. Luego utiliza la misma para experimentar sobre los límites del funcionamiento de `Ethereum` cuando se reduce el tiempo entre bloques.

Kai Otsuki et al. [28] analizan los efectos de las redes de retransmisión, utilizando SimBlock, un simulador de redes blockchain. Realizando una simplificación del modelo de red de retransmisión, hacen simulaciones con 6000 clientes de Bitcoin, y analizan el tiempo de propagación y la cantidad de *forks* variando la relación entre clientes conectados a la red de retransmisión.

Nicolas DeCarli [29] analiza la viabilidad de modificar el *target time* de la red de Bitcoin, para mejorar el *throughput* del sistema. Utilizando `SherlockFog`, realiza experimentos con una red emulada de Bitcoin de hasta 480 clientes.

Julio Augusto Mascitti [30] estudia el impacto de la implementación de redes de retransmisión en Bitcoin como alternativa para mejorar el desempeño del sistema y así evitar los cuellos de botella que presenta para manejar grandes volúmenes de transacciones. En su trabajo, emula una red de Bitcoin utilizando `SherlockFog` y así intentar representar las condiciones de la red real.

En este subconjunto de trabajos, encontramos una motivación para continuar el análisis de redes de *blockchain*. Cambiando el foco a las redes de *Proof of Stake*, tomamos muchas de las ideas en cuanto a emulación de la red y de las magnitudes a analizar.

En cuanto a temas teóricos, vimos el trabajo de Lewis-Pye et al. [31] donde realiza un análisis análogo al teorema CAP, comparando las redes de PoW y PoS en cuanto a su adaptación a distintos niveles de participación en el consenso y su finalidad (es decir, su capacidad para completar una ronda). Discute que ningún protocolo es capaz de cumplir ambas cosas a la vez, donde PoW es un ejemplo de red que se adapta fácilmente a cambios en la cantidad de nodos participando de la red, pero no asegura que llegará al consenso. A diferencia de esto, PoS tiene parámetros en cuanto a lo que necesita para llegar al consenso, siendo más limitante en cuanto a la adaptación a cambios en la participación, pero que aseguran que las rondas se llevarán a cabo. Otro dato interesante del trabajo es que compara el mecanismo de comunicación de la red en cuanto a las decisiones: mientras que en PoW primero se elige el bloque y luego se propaga por la red, en PoS sucede de la manera inversa, dado que todos los nodos propagan los bloques propuestos y finalmente bajo un criterio se elige uno de ellos.

Por otro lado, vimos en Lepore et al. [32] una comparación entre PoW, PoS y $PPoS$ donde se analizan los mecanismos de consenso, su finalidad, latencia, *throughput* y escalabilidad. En estos casos, PoW no presenta finalidad, su latencia es lenta (hay que esperar varios bloques para considerarlo confirmado), acepta pocas transacciones por segundo, pero escala fuertemente en la cantidad de mineros. En cambio, en PoS y $PPoS$ contamos con finalidad, latencia mucho menor y soporte para miles de transacciones por segundo. El problema en ambas es su escalabilidad ante el crecimiento de las cuentas participantes en el consenso. En PoS puede volverse complejo en base a como se toman las decisiones de la red (si hay un validador que elige o si se utiliza una estrategia de cadena más larga para hacerlo), mientras que en $PPoS$ puede afectarse el tamaño de los comités para afrontarlo.

También analizamos una comparación por dentro de las redes que implementan PoS en Jiseong et al. [33] para conocer cómo se encuentra `Algorand` a nivel de distintas propiedades. En este trabajo se toman múltiples de estas redes (que incluyen a `Algorand`, `Avalanche`, `BNB`, `Cosmos`, `Ethereum`, `Solana`, `Polygon`, entre otras) y crean cinco parámetros de comparación entre ellas: cantidad de validadores, capital necesario para participar, concentración de capital, costos operacionales, estabilidad de la red en términos económi-

cos (*tokenomics*). Con estos parámetros intentan ver qué tan sincera o transparente es la red (busca medir la *openess* de ellas). Si bien `Algorand` se encuentra en el segundo lugar de mejores resultados para esta propiedad (siendo superada solo por `Avalanche` y encontrándose cercana a `Ethereum` y `Solana`), nos brinda un buen criterio para poder comparar redes de *PoS*. En Mogavero et al. [34] se genera una comparación similar, pero agregando un nuevo valor al trilema: el costo de ejecutar código de manera descentralizada (convirtiéndolo así en un *quadrilemma*), viendo que `Algorand` resulta una buena opción para cómputos simples debido a su bajo costo, pero limitado por el lenguaje que utilizan para los contratos inteligentes, siendo superado por `Ethereum` cuando son cálculos más sofisticados, aunque con un costo muy superior, pero que ninguna de las redes termina de cumplir con tener transacciones con bajo costo y tiempos de confirmación rápidos.

Con esta base, tomamos otras investigaciones más cercanas a `Algorand`. Más allá de los trabajos de análisis económico de la red (como en los trabajos de Nicola Dimitri [35,36], donde revisa la manera en que `Algorand` premia a la red por su participación y cuál podría ser el *stake* óptimo de un nodo participante para maximizar sus ingresos), nos enfocamos en estudios teóricos y prácticos sobre distintas características de la red.

En primer lugar, vimos un trabajo teórico donde se critica la falta de un mecanismo de control para usuarios maliciosos y donde se propone agregar un sistema de puntuación para las cuentas y así incorporarlo en la toma de decisiones, quitándole poder de voto a aquellos que actúan de mala manera (trabajo de Pandey et al. [37]). En otro trabajo teórico, Gouget et al. [38] investiga la impredecibilidad de las decisiones de la red (no queremos que las mismas puedan ser predichas), encontrando que `Algorand` lo cumple, pero que no resulta *t-forward* impredecible. Es decir, que tomando todo lo conocido de la red hasta un cierto punto, tenemos información suficiente como para predecir desde ese punto cuáles serán las elecciones de líder (por ejemplo). Sin embargo, no analiza la distribución del *stake* como factor en la toma de decisiones, sino que solo revisa la aleatoriedad de las elecciones para una red distribuida de manera equitativa.

Luego, analizamos dos trabajos donde se trata el hecho de que `Algorand` no entrega premios a quienes procesan las transacciones del sistema: no hay ningún tipo de beneficio para los nodos por hacerlo, sino que los costos que se le cobran al emisor de la misma (es decir, el *fee*) se guarda en una cuenta de `Algorand` para su uso futuro. En el trabajo de Abbasihafshejani et al. [39] se plantea el hecho de que para los nodos es más rentable generar bloques vacíos (por lo cual pueden recibir una recompensa) en lugar de tomar transacciones y tener que utilizar poder de cómputo para procesarlas, teniendo así un gasto extra por el que no reciben nada a cambio. Proponen modificar el sistema de recompensas para así fomentar todas las acciones de la red y evitar estas decisiones egoístas de los nodos. Para esto, buscan un tamaño óptimo de bloque que evite que los nodos lo generen vacío y que no lo sobrecarguen para tener rédito rápido por el costo de procesamiento de las transacciones. También analizan la relación entre el tamaño del bloque con su tiempo de propagación, utilizando una herramienta de simulación de red de Bitcoin (SimBlock), pero extrapolando los resultados para las propiedades de `Algorand` (es decir, no implementan una herramienta propia para simular una red de `Algorand`).

El mismo tema es revisado por Fooladgar et al. [40], donde se mide el costo de cada operación de la red (sea verificación de transacciones, generación de bloques, ejecución de la *VRF*, etc.) para entender la rentabilidad de tener un nodo. Aquí, los autores utilizan un simulador propio para correr una red privada pudiendo configurar distintos tamaños de red y latencias entre los nodos, aplican una distribución uniforme del *stake* (pudiendo también

cambiarlo para ver el efecto) y en sus experimentos detectan que si el 15 % de los nodos deciden no participar del consenso por esta falta de recompensas, la red empieza a fallar y tiene problemas para lograr consenso. Realizando un ajuste en la entrega de recompensas, encuentran un mejor esquema que fomenta la participación de todos los nodos en todas las interacciones de la red.

Este último trabajo nos resulta de sumo interés debido al esquema que plantea para sus experimentos: instancias privadas de redes de `Algorand` donde podemos emular una red física modificando latencias y revisando el efecto que tiene afectar una porción del *stake* en cuanto al avance del protocolo de consenso.

Por otro lado, buscamos trabajos sobre ataques en `Algorand`, habiendo visto similares para `Ethereum` en su versión *PoS* [41].

El trabajo de Conti et al. [7] es el primero en hacer un análisis de seguridad sobre `Algorand`, buscando una manera de afectar a nodos de la red para que no puedan participar del consenso y así potencialmente evitar que avance. El ataque consiste en utilizar los mensajes que se envían entre los nodos para llenar la memoria del nodo objetivo, aprovechando que el protocolo de consenso debe verificar cada mensaje recibido y si no corresponde a la ronda actual (es decir, es de una ronda posterior) debe almacenarlo hasta que llegue esa ronda. Entonces, enviándole muchos mensajes que no pueda procesar aún puede saturarse su espacio en memoria y así lograr degradarlo. Para esto eligen los mensajes de propuesta de bloque que son los más grandes que maneja el sistema. Desarrollan su propia herramienta para simular una red de nodos (aunque no experimentan con latencias en sus enlaces) y trabajan con redes de 500 nodos, donde van variando la cantidad de nodos maliciosos y cantidad de cuentas maliciosas en cada uno. Este último factor tiene que ver con que el nodo objetivo puede reconocer que está recibiendo muchos mensajes de una cierta cuenta o nodo y potencialmente podría bloquearlo y descartarlo, con lo cual es necesario distribuir el ataque para que sea más indetectable. Con distintas configuraciones logran que la creación de bloques se demore hasta 390 s (cuando el tiempo normal ronda los 4 s), pudiendo acelerar el efecto en el nodo si se incrementan la cantidad de cuentas que lo atacan, pero con la contrapartida de que el nodo va a bloquearlas eventualmente y necesitaríamos nuevas para seguir el ataque.

En Wang et al. [42] y Lee et al. [43] se analizan también dos ataques teóricos de índole económica, donde el primero detecta la posibilidad de crear un nuevo *fork* de la red a partir de una cuenta con menos de un tercio del *stake* (algo que contradice la propiedad de imposibilidad de *forks* en `Algorand`), mientras que en el segundo se analiza un ataque de venta en corto (*short selling*) en el que un usuario utiliza un préstamo de *algos* para afectar el consenso, afectar la red, potencialmente bajar el precio de la moneda y así ganar dinero.

Nos resulta de sumo interés el trabajo de Conti et al. ya que plantea un sistema simulado donde analiza cómo se modifica el tiempo de ronda del sistema en base a ataques en los nodos.

Con respecto al análisis de la información, decidimos aplicar una metodología clásica como es el análisis de ventanas por desplazamiento en lugar de enfoques basados en grafos dinámicos [44, 45]. Esta decisión se basó, por un lado, en la gran complejidad y recursos necesarios para el segundo enfoque, pero principalmente debido a que la estructura de la red de `Algorand` no cambia significativamente durante su ejecución (basado en lo visto a partir de documentación y código). Si bien `Algorand` realiza mediciones para analizar enlaces y reemplazarlos ante problemas de desempeño, no pudimos encontrar fuentes ni

implementaciones que fundamenten un cambio de conexiones de manera arbitraria. El enfoque de grafos dinámicos, si bien puede cubrir este escenario estático, nos da herramientas para hacer análisis de sistemas cambiantes y así obtener información certera sobre ellos: podemos crear grafos donde los enlaces se van modificando en el tiempo, agregar pesos en los mismos, agregar nodos, y muchas otras opciones que captan de mejor manera un sistema vivo, con la contrapartida de mayores requisitos computacionales para mantener esta representación.

1.5.1. Objetivos

En base a lo visto en los trabajos existentes notamos una falta en la exploración de cómo se comporta la red ante modificaciones o ataques en la topología física: qué sucede si se cortan los enlaces entre los nodos o los *relays*, o si aparecen latencias que agregan tiempo en la comunicación entre ellos. Estos podrían ser vectores de ataque si afectaran una cantidad importante del *stake* activo que participa del consenso, pudiendo hasta detener el avance de la red. Esto último se debe a que `Algorand` encara el teorema CAP de manera de no funcionar ante particiones que afecten una porción grande del *stake*, enfocándose en tener un buen tiempo de recuperación.

Esta exploración será el primer objetivo de la tesis, viendo cómo se comporta `Algorand` bajo distintas topologías físicas que reciben distintos tipos de afecciones en sus enlaces. Para esto trabajaremos con redes físicas emuladas donde podamos ejecutar una instancia privada de `Algorand`, evitando que la misma corra en enlaces por fuera de nuestro control (por ejemplo, en Internet).

Por otro lado, la arquitectura de la red logra anonimizar a los nodos de la red: los *relays* son públicos y puede obtenerse un listado de ellos fácilmente, pero los nodos que manejan las decisiones con las cuentas que manejan se encuentran ocultos detrás de ellos. Sólo los *relays* a quienes están conectados podrían conocer su IP y asociarla a las cuentas que manejan en base a los mensajes que reciben del mismo: por ejemplo, si tuviéramos control de un *relay* y analizamos los mensajes de propuesta de bloque que envía un nodo, podríamos suponer que la cuenta que firma estos mensajes se encuentra administrada por él. También, si este *relay* se encarga de enviar estos mensajes a los demás *relays*, debería ser uno de los primeros en hacerlo o el que más mensajes envía de esa cuenta.

Con esta estrategia, nos podemos poner en lugar de un *relay* afectado que, maliciosamente o sin saberlo, se encuentra analizando los mensajes que recibe tanto de nodos como de *relays* para poder encontrar las cuentas más importantes o a qué *relays* podrían estar conectados.

El segundo objetivo del trabajo será entonces encontrar una manera de poder ubicar a los nodos dentro de la topología lógica del sistema **solamente** en base a los mensajes recibidos por un *relay* de la red. Por sobre todo, nos resultará de sumo interés encontrar a las cuentas con mayor *stake*.

2. METODOLOGÍA

En esta sección se describen las herramientas desarrolladas y utilizadas para este trabajo y los experimentos realizados. Si bien se da una explicación detallada, se puede encontrar un mayor nivel de detalle en el paquete de código de esta tesis.

2.1. Introducción

En este trabajo nos propusimos trabajar con instancias privadas de la red de `Algorand` en ambientes controlados, de manera de tener control total de la misma, no solo a nivel lógico, sino también a nivel físico: si utilizáramos nodos distribuidos en *Internet* perderíamos ese control dada la impredicibilidad de sus enlaces. Nos interesó especialmente que los nodos operen tal como lo harían en la red real, queriendo así emular la misma (y no simularla como en otros trabajos analizados). También, buscamos construir sistemas de diferentes tamaños, llegando a la cantidad de nodos de la red real, con lo cual la escalabilidad de nuestras herramientas fue crítica.

Para lograr el control del aspecto lógico, debimos desarrollar herramientas que nos permitan crear configuraciones de nodos de forma que funcionen en una red privada, logrando que progrese. Así, evitamos que intenten conectarse a los *relays* curados por la fundación `Algorand`, utilizando los de nuestra instancia.

Con respecto al aspecto físico, utilizamos **SherlockFog**, una herramienta que nos permite emular redes físicas sobre un conjunto de servidores. Así, podemos crear nodos que van a correr la red lógica de `Algorand` sobre una capa física emulada en la cual podemos crear y controlar todos los enlaces de la misma, pudiendo además modificar las latencias de todos ellos durante la emulación.

Finalmente, en cuanto a la escalabilidad, si bien a nivel lógico solo implicó ajustar configuraciones en los nodos, pudimos crear redes físicas de gran tamaño gracias a la posibilidad de ejecutar **SherlockFog** sobre múltiples servidores. De esta manera, podemos distribuir la carga de los sistemas más grandes para evitar que la *performance* se degrade y afecte el normal funcionamiento de la red.

Si bien **SherlockFog** nos permitió distribuir esta carga, necesitamos encontrar servidores sobre los cuales hacerlo. Para esto hicimos uso de **CloudLab**, un conjunto de servidores físicos dedicados para investigación a los que podemos acceder a nivel *bare metal* para ejecutar nuestros sistemas.

2.1.1. Cómo definimos un experimento

Además de estos desafíos técnicos, nos interesa definir qué es un experimento para este trabajo. Si bien en algunos casos vamos a agregar consideraciones específicas que definiremos en cada uno, podemos establecer una base inicial para cuál es el procedimiento de creación de los mismos.

1. Determinar el objetivo del experimento
    - Definimos qué es lo que queremos analizar (qué pregunta buscamos responder), qué duración debería tener el experimento, y qué información debemos extraer

del sistema (debiendo modificar el cliente de ser necesario para registrarla en los *logs* de cada nodo).

2. Definir su escala y distribución de *stake*
   - Buscamos qué tamaño de red necesitamos para cumplir nuestro objetivo, y cómo debemos distribuir el *stake* en la misma.
3. Elegir las configuraciones de nodos necesarias a nivel lógico
   - Cómo se deben establecer las conexiones o el *stake* de cada nodo y *relay* (entre otras configuraciones).
4. Definir la arquitectura de la red física
   - Definimos los nodos y enlaces de la red física a emular con **SherlockFog**.
5. Decidir las acciones lógicas durante la ejecución
   - Ya sea inyectar transacciones u otras acciones a nivel de la red de `Algorand`.
6. Definir las alteraciones físicas durante la ejecución
   - Por ejemplo, modificar las latencias de los enlaces físicos.
7. Seleccionar el ambiente donde ejecutar el sistema
   - Dependiendo del tamaño y las acciones a realizar, utilizamos un ambiente local para pruebas pequeñas o sencillas, y **CloudLab** para más grandes o complejas.
8. Verificaciones de la ejecución
   - Definimos qué validaciones realizar sobre el sistema ya sea durante o luego de la ejecución.
   - Por ejemplo, utilizamos el tiempo por bloque para medir la salud del sistema durante el experimento, y analizamos los mensajes enviados para verificar la participación de cada nodo al finalizar el mismo.
9. Análisis de los datos registrados
   - En base a los objetivos planteados y los datos registrados, realizamos el análisis de los mismos para obtener una respuesta a nuestra pregunta inicial.
   - Para esto utilizamos los *logs* generados por cada nodo al finalizar la ejecución.

Con estos pasos establecemos un procedimiento básico para definir cada experimento. Veamos entonces de manera detallada las implicancias técnicas de este proceso.

2.2. Adaptación del cliente

Si bien para este trabajo no se realizaron modificaciones significativas en la operación del cliente, el mismo fue modificado para poder capturar y *loggear* todos los eventos que resultaban de interés para el trabajo. Se tomó el cliente original desarrollado en GO, disponible en los repositorios de `Algorand`, en su versión 3.0 [46] y se creó un nuevo módulo que se encarga de *loggear* en un archivo específico de eventos. Todos estos eventos se *loggean* con fecha, hora y un identificador que nos permite luego hacer un análisis de los mismos a través de *scripts*.

A su vez, también se modificó el cliente para que no verifique las transacciones recibidas de manera de agilizar ese proceso en la emulación y evitar sobrecargar el sistema con este chequeo: si bien en un nodo real puede resultar insignificante el costo de la verificación, como nosotros ejecutamos múltiples de estos nodos en el sistema y hacemos pruebas inyectando miles de transacciones en un momento dado, preferimos no validarlas y enfocarnos en el proceso de la red más allá de la verificación.

Los *logs* específicos para la tesis se administran por la estructura `ThesisBundle` ubicado en `go-algorand/thesis/thesis.go`. Este paquete se puede inyectar en los lugares donde quieran generarse *logs*, utilizando las funciones provistas.

El *log* se inicializa en el directorio indicado por la variable de entorno del sistema configurada para `ALGORAND_THESIS_LOG_PATH`, agregando al final del archivo cada uno de los *logs* recibidos en formato *JSON*. Entre los campos del mismo contamos con el tipo de *log* generado (para poder luego identificar a qué tipo de evento se corresponde, por ejemplo para el conteo de tipo de mensajes o para mensajes de un tipo específico manejados por el cliente), el día y horario del evento, quién envía el mensaje (por ejemplo, para identificar de qué *relay* provino), a qué cuenta hace referencia. Estos campos pueden variar según el tipo de mensaje, lo cual nos permite ser flexibles en cuanto a los datos a guardar de cada evento, pero almacenarlos en el mismo archivo final.

Los tipos de *logs* pueden habilitarse al comienzo del archivo configurando los *flags*:

```
var printTxLogs = false
var printBlockLogs = false
var printSpecificMessageLogs = false
var printIncomingMessageLogs = false
var printDisconnectMessageLogs = false
```

En orden, se corresponden con *logs* para las transacciones procesadas, los bloques procesados, mensajes con tipos específicos, mensajes de cualquier tipo, y eventos de desconexión del cliente hacia otros nodos.

Luego de configurarlos, deben volver a compilarse los binarios. Con esto podemos predefinir qué conjunto de registros queremos tomar de un experimento, pudiendo dejar todo habilitado, pero con el costo de *logs* mucho más grandes. Cada evento cuenta con una función específica que recibe todos los datos a guardar en el *log*, genera internamente el *JSON* correspondiente y finalmente lo agrega al final del archivo de *log*.

Con este enfoque, implementamos un módulo aislado dentro del cliente que podemos inyectar en los distintos componentes del mismo para monitorear eventos sin alterar el comportamiento original (solo incluyendo llamadas a nuestro módulo para generar estos *logs*).

2.2.1. Eventos registrados

Con fines de monitorear el avance de la red y las mediciones que nos interesan registrar de la misma, manejamos un conjunto de eventos que los podemos categorizar de la siguiente manera:

- Eventos de bloques:
    - Bloque propuesto por el nodo para la ronda actual.

    - Bloque aceptado en la ronda por la red.

- Eventos de transacciones:

    - Transacción creada en el nodo.
    - Transacción recibida desde otro nodo.
    - Transacción agregada a bloque.

- Eventos de mensajes:

    - Mensaje recibido de un tipo específico con detalle.
    - Tipos de mensajes recibidos.
    - Mensajes de desconexión de otros nodos.

Este conjunto de eventos puede ser extendido fácilmente llamando a las funciones que los manejan desde cualquier parte del cliente.

2.3. Imagen `docker` para el cliente

En nuestros experimentos nos interesa ejecutar múltiples clientes de la red `Algorand` para poder emular distintos escenarios. Para poder lograrlo, necesitamos virtualizar estos nodos de manera de poder escalarlos y manejarlos en un conjunto limitado de servidores de manera controlada.

`Algorand` provee documentación no solo para generar imágenes `docker` de su cliente, sino también herramientas para emular redes privadas para hacer experimentos, pero las mismas no permiten hacer modificaciones en la capa física ni personalizar el contenido o comportamiento del cliente normal (excepto modificando las plantillas provistas de `docker`). Si bien son herramientas útiles para hacer pruebas en ambientes privados (por ejemplo, para testear aplicaciones o *smart contracts*), no resultan convenientes para experimentos donde nos interese poder configurar desde cero la topología física y lógica de la red.

Por estos motivos, desarrollamos una imagen `docker` genérica del cliente de `Algorand` que nos permita instanciarla como tantas veces necesitemos, pudiendo establecer sobre la marcha las configuraciones que deseemos para los experimentos. Esta imagen cuenta con todos los binarios compilados de la sección anterior, de manera que corre nuestra versión modificada del cliente. También incluimos *scripts* que permiten ejecutar de manera externa las configuraciones y ejecución del cliente:

- `01-copyNodeFolder.sh`: recibe el *path* de una configuración de nodo y se encarga de copiarlo al *path* donde se encuentran los binarios de `Algorand`.
- `02-startNode.sh`: corre la instancia del cliente, eligiendo la configuración copiada en el paso anterior para la ejecución.
- `03-stopNode.sh`: detiene la ejecución del cliente.
- `04-copyPostMortem.sh`: copia toda la carpeta de configuración con los *logs* generados por el cliente a una ruta configurada por parámetro.

Con estos *scripts* podemos tener imágenes genéricas y configurarlas cuando se corre el experimento, manejando un primer paso de configuración, luego se corren los clientes y finalmente detenemos el experimento cuando ya terminó y recuperamos los *logs* del cliente. Estos *logs* son los propios de `Algorand`, no incluyen nuestros *logs* para la tesis, pero nos permiten revisar cualquier situación que se desvíe de lo normal.

Los *logs* de la tesis se guardan en una ruta configurada al inicio del experimento como variable de entorno, con el nombre del nodo que lo genera, quedando todos ellos en un solo lugar compartido y externo a las instancias. Así, si el experimento se detuviera por algún motivo, mantenemos los *logs*.

Aprovechando que generamos una plantilla para este cliente, también agregamos utilidades para administrar y revisar configuraciones de red para poder hacer pruebas de conectividad internas y asegurarnos de que las conexiones entre los nodos estén funcionando correctamente.

2.4. Configuraciones de nodos

La imagen `docker` que generamos es puramente genérica y no contiene ninguna configuración que le permita ejecutar una instancia de un cliente de `Algorand`, más allá de contener los binarios necesarios para eso. Necesitamos, entonces, generar las configuraciones de nodo para que los clientes funcionen. Algo muy importante en este punto es que para poder garantizar la repetitibilidad de los experimentos, las configuraciones deben ser las suficientes como para que no quede nada librado al azar y podamos obtener los mismos resultados con un conjunto de ellas (o muy similares, dado que aún así hay factores que siguen siendo aleatorios en la ejecución del sistema, pero que no dependen de la configuración).

En nuestros experimentos, queremos que cada cliente esté asociado a una única cuenta del sistema. Por esto, contamos con un *script* de `Python` que se encarga de generar una cantidad base de nodos, que van a crearse en carpetas separadas por cada nodo en un *path* indicado, con las siguientes configuraciones:

- Nombre de nodo (siguiendo la sintaxis `n<número>`, como “n15”) y su carpeta.
- *Token* de administración del nodo: lo necesitaremos para poder ejecutar comandos en el nodo externamente.
- Archivo génesis inicial (luego debe ser modificado para la red a emular).
- Creación de una billetera o cuenta con una contraseña por defecto configurada en el ambiente como variable de entorno, generando sus claves públicas y privadas.
- Creación de una clave de participación de esa cuenta en la red.
- Asociación de esa clave al nodo, para permitirle representar a la cuenta en el sistema.

Con estos puntos generamos un nodo genérico con una cuenta asociada que participa del consenso a través de su cliente. Ninguno de estos valores están asociados a una red, sino que luego podemos utilizarlos como base para crear los nodos que van a participar de cada experimento. Podemos utilizar todos o un subconjunto de estos nodos base generados, con lo cual normalmente creamos una gran cantidad que alcance para todos los tamaños de experimentos que necesitamos (pudiendo luego volver a generarlos si se requieren más).

Resulta importante remarcar que algunos de los pasos utilizados no son reproducibles solo con el SDK provisto por `Algorand` para `Python`, de manera que requerimos de los binarios del cliente para poder completar esta generación, teniendo que correr brevemente el nodo con esta configuración y así lograrlo.

A medida que se crean los nodos, vamos también almacenando los valores de las configuraciones que podemos necesitar posteriormente de cada uno, como el *token* de administración del nodo, las claves públicas y privadas de la cuenta creada para este, entre otros.

2.5. Creación de la red para el experimento

Ya contamos entonces con un conjunto de nodos base que nos van a servir para crear una red, pero por el momento sus cuentas no contienen *stake*, ni conocen a los demás nodos y *relays*. Con este fin contamos con varios *scripts* que se encargan de crear toda la red en sí, entre ellos `createNetwork-line-equal-rxn.py` y `createNetwork-line-realistic-rxn.py`. Ambos reciben como parámetro la cantidad de nodos que queremos en la red, la cantidad de *relays*, la cantidad de *relays* conectados por nodo y un directorio para generar la carpeta con toda la salida de la creación. La diferencia entre ellos es que el primero genera una distribución homogénea del *stake*, mientras que el segundo sigue una distribución que respeta la real de la red actual de `Algorand`, obteniendo una distribución heterogénea donde algunas cuentas mantienen una porción mucho más grande del *stake* que otras.

Dentro de este proceso, encontramos tres partes fundamentales:

- Creación de genesis.json: contiene el génesis de la red, donde se asocia un *stake* inicial a cada una de las cuentas participantes del consenso, sus claves públicas de participación en el consenso (que son las que deben configurarse en los nodos que van a correr esa cuenta), el nombre de la cuenta, la versión del protocolo de consenso que utilizará la red, entre otros valores.

- Configuración de nodos y *relays*: tomamos una configuración base para los nodos y otra para los *relays*, copiamos la cantidad de nodos base necesarios para el experimento y configuramos el *stake* de cada nodo (según lo elegido como parámetro, pudiendo ser homogéneo o heterogéneo) y los *relays* conocidos para cada uno, guardando esta información en otro `csv` para tener trazabilidad de esta arquitectura.

- Creación de genesis hash: finalmente, corremos una instancia de un cliente con la configuración dada previamente para calcular el hash del génesis creado, que luego será necesario para poder inyectar transacciones en la red.

Con este paso, ya creamos una configuración de red para poder ser utilizada con los contenedores del paso anterior, contando con un `csv` detallando cada uno de los nodos con su *stake* y sus conexiones en la red, lo cual nos facilita luego el análisis. Todo esto se genera en una carpeta que contiene cada uno de las configuraciones de los nodos y que pueden utilizarse para los clientes `docker`. Estos clientes copiarán una de las carpetas de configuración (por ejemplo, la del nodo "n22") y la utiliza cuando inicia el proceso principal del cliente.

2.5.1. Creación e inyección de transacciones

Más allá de experimentar con distintas configuraciones de red lógica y física, también nos interesa tener la capacidad de inyectar transacciones en el sistema durante la ejecución del experimento. Para lograr esto de manera repetible, dada una red experimental generamos todas las transacciones que queremos inyectar previo a la ejecución del experimento.

El primer paso es crear las transacciones de forma *offline*, firmarlas con la clave privada del nodo que la genera y guardarla en un archivo que contenga toda esta información. Para este paso contamos con dos opciones: generar transacciones por separado o generar grupos de transacciones que serán enviadas en conjunto (con hasta 16 transacciones en el mismo). La primera opción nos permite tener un control fino de qué transacción se envía en cada momento, pero con mayor costo (ya que debe enviarse una por una). La segunda opción nos permite mandar una gran cantidad de transacciones en cada envío, pudiendo lograr una mayor cantidad por unidad de tiempo sin incurrir en el esfuerzo que requeriría hacerlo una por una. En ambos casos las transacciones pueden crearse con anterioridad a que la red se ejecute.

Contamos con *scripts* para generar tanto transacciones individuales como agrupadas de manera sencilla (`createTx.py` y `createTxGroup.py` respectivamente), considerando los parámetros que requiere `Algorand` para las mismas, entre los que se encuentran:

- Emisor: cuenta de la cual salen los fondos de la transacción.
- Receptor: cuenta que los recibe.
- Cantidad: la cantidad de esos fondos que deben transferirse.
- Versión del consenso utilizada: hace referencia a qué versión del algoritmo de consenso se utilizó al crear la transacción, debiendo ser el que utiliza la red (valor que se agrega para contemplar la escalabilidad de este algoritmo).
- Hash del génesis de la red: indica para qué red fue creada la transacción, logrando que solo sea válida en ella.
- Rango de rondas donde es válida: se puede especificar a partir de qué ronda y hasta qué otra la transacción puede procesarse. Si un nodo la recibe antes de tiempo, puede almacenarla para procesarla luego. Pero si la recibe cuando ya pasó ese rango, es descartada. Este valor nos permite predefinir en qué momento debe procesarse cada una.
- *Fee*: es el valor a pagar por el procesamiento de la transacción (*gas*). Si no es el suficiente (sea porque es muy bajo inicialmente o porque la red se encuentra recibiendo una gran cantidad de trabajo), la transacción puede no llegar a procesarse, debiendo tener que regenerarla.

Así, podemos crear todas las transacciones que querramos para el experimento para un momento dado y dejarlas listas para ser inyectadas.

Entonces, la segunda parte es justamente enviarlas al sistema. Contamos por un lado con un *script* que dado un archivo de transacción y un nodo a quién enviársela (que podría no ser el nodo asociado a la cuenta que genera la transferencia), la envía y guarda el mensaje de resultado de la misma. Este resultado nos indica si la misma fue aceptada por

el nodo para su procesamiento o si hubo algún problema con el mismo (por ejemplo, si el *fee* no fue suficiente). Por otro lado, creamos un *script* que dado un conjunto de transacciones, se encarga de invocar al *script* de envío cuando llega la ronda que le corresponde a cada una. Para lograr esto, el *script* consulta periódicamente a un nodo por el valor de ronda actual y en base al tiempo promedio de ronda aguarda un tiempo similar al que debería tardar en llegar a la ronda objetivo. Cuando se encuentra cerca, aumenta el *rate* con el que consulta por ese valor y al alcanzarlo, envía las transacciones de esa ronda.

Para el envío de las transacciones utilizamos la `API Rest` de cada nodo, autenticándonos con el *token* de cada uno de ellos, guardado al generarlos.

2.6. SherlockFog

En nuestros experimentos, nos va a interesar poder emular distintas topologías físicas de red sobre la cual va a correr la red lógica de `Algorand`. Para esto requerimos una herramienta que nos permita crear distintas configuraciones, ejecutar experimentos y tener la flexibilidad para alterar las latencias entre los distintos nodos del sistema. Con este fin, vamos a utilizar la herramienta `SherlockFog`.

En Geier et al. [47] se introduce el lenguaje *fog*, que es utilizado para describir una topología de red IP y definir comandos a ejecutar en los *hosts* de dicha red. En el mismo trabajo se presenta `SherlockFog`, una aplicación que toma un *script* escrito en *fog* y ejecuta comandos de *shell* que logran una emulación de la topología deseada y ejecutan en los *hosts* virtuales lo especificado en el *script*. Cada nodo de la red generada es emulado mediante un contenedor de Linux que es un ambiente aislado adecuado para realizar experimentos. Utilizamos `SherlockFog` para emular la red IP y correr, de manera automatizada, los módulos necesarios para inicializar los experimentos.

En la figura 2.1 podemos ver una esquematización de cómo funciona `SherlockFog`. Podemos diferenciar los siguientes pasos:

1. El usuario elige una aplicación y una topología y construye un *script* que define un experimento en `SherlockFog`. Dicho *script* utilizará un conjunto de nodos físicos especificados por el usuario.

2. `SherlockFog` es ejecutado en el coordinador, el cual se conectará a cada nodo para inicializar la red virtual.

3. Se generan enlaces virtuales correspondientes a la topología descripta en el *script* de entrada. Se utiliza ruteo estático para permitir la comunicación entre cada interfaz de red virtual.

4. El código de cada aplicación es ejecutado en los nodos virtuales según lo que haya especificado el usuario en el *script* de entrada.

`SherlockFog` permite modificar los parámetros en tiempo de ejecución, y repetir los experimentos con distintos parámetros o topologías. La salida puede ser recolectada para su posterior análisis.

Por las siguientes consideraciones se decidió utilizar `SherlockFog` para llevar adelante los experimentos de esta tesis.

- `SherlockFog` puede ser ejecutado en hardware convencional.

- Los *script*s son escritos en un lenguaje propio denominado *fog*, que permite definir la topología de red, parámetros de los experimentos y comandos para ejecutar la o las aplicaciones.
- La herramienta puede ser utilizada sobre un único nodo físico o sobre varios, permitiendo escalar la red emulada sobre otros recursos.
- La red, CPU y la memoria pueden ser compartimentadas entre nodos virtuales que se instancian sobre un mismo sistema utilizando técnicas de virtualización liviana.
- El código de usuario se puede ejecutar sin ningún tipo de modificación, permitiendo la evaluación de programas tanto de código abierto como cerrado.
- La dinámica del sistema de comunicaciones puede ser modelada cambiando el ancho de banda o la pérdida de paquetes de un enlace en tiempo de ejecución.

Luego, para cada uno de los experimentos que querramos plantear creamos un *script* con formato *fog* que representará la topología física del sistema y los otros *scripts* a ejecutar durante el mismo, ya sea esperar un cierto tiempo o una cierta cantidad de rondas para afectar un enlace o inyectar transacciones en la red.

Tomando la configuración del archivo en formato *fog*, se levantan distintos contenedores en varios servidores físicos y se establece la topología física con las latencias que pretendemos. Así, en cada uno de los hosts vamos a poder ejecutar un cliente `Algorand` con la configuración predefinida.

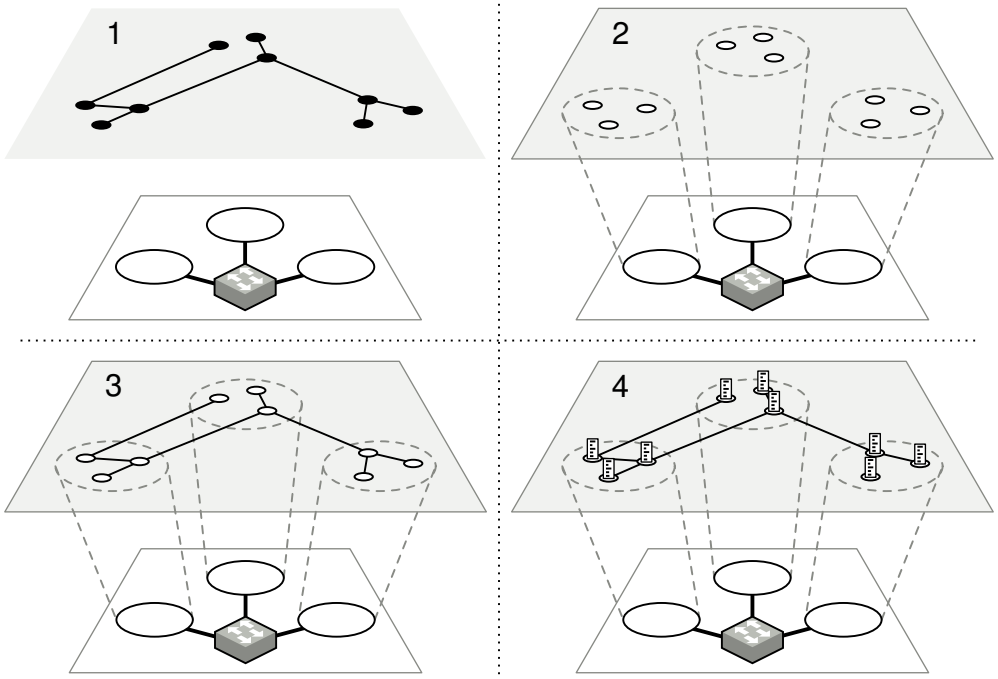


Fig. 2.1: 1) La definición en lenguaje *fog* del experimento a realizar. 2) La creación de los contenedores necesarios. 3) La emulación de la red lógica pretendida. 4) La ejecución de distintos *scripts* en cada nodo de la red. `SherlockFog` permite ejecutar varios clientes `Algorand` (ya sea en el mismo host físico como en distintos), teniendo control de la red que los comunica.

2.6.1. Creación del *script* para `SherlockFog`

En esta instancia ya contamos con un conjunto de nodos y *relays* listos para correr en una red emulada, pero queremos poder hacerlo de manera de poder definir, además, la topología física sobre la que corre la misma. La idea será poder configurar ciertas conexiones entre los nodos que puedan modificarse a lo largo de un experimento y así realizar pruebas sobre el comportamiento de la red en esas condiciones.

Para poder emular una topología de red física vamos a utilizar `SherlockFog` que nos permite definir una cierta cantidad de nodos, los enlaces entre ellos (junto con sus latencias), y la posibilidad de ejecutar comandos sobre la red creada. Esta herramienta toma una imagen `docker` y se encarga de generar todo lo necesario para poder emular una topología física, corriendo nodos con la imagen elegida. Para ello, se puede configurar una instancia de `SherlockFog` a través de su propio lenguaje, creando una plantilla con todo lo que queremos crear en el experimento.

Por ejemplo, con la sentencia `def n1` y `def n2` definimos dos nodos, n1 y n2, para la red. Luego, podemos conectarlos utilizando `connect n1 n2 20ms`, lo cual crea un enlace físico entre ambos nodos con una latencia de 20 ms (nuevamente, este enlace es emulado, no es un enlace físico real). Una vez creada la topología física, es decir, que se definieron todos los nodos y los enlaces que se requieren para el experimento, se utiliza la sentencia `build-network` para crear esta red. A partir de este punto, ya contamos con contenedores `docker` corriendo para cada nodo, con una imagen preseleccionada al momento de correr la herramienta. Podemos correr comandos en alguno de los nodos utilizando `run n1 bash /root/go/bin/01-copyNodeFolder.sh -n n1`, comando que se encarga de correr el *script* que copia la carpeta de configuración de n1 al directorio necesario para correr el cliente de `Algorand`.

Otro beneficio de `SherlockFog` es que podemos afectar los enlaces durante el experimento y modificar su latencia, pudiendo agregar *delays* en los mismos. Esto lo logramos con la sentencia `set-delay n1 n2 1000ms`, que en este caso configura un *delay* de 1000 ms entre los nodos n1 y n2. Esta funcionalidad es de suma importancia, porque nos va a resultar de gran interés experimentar acerca de cómo se adapta la red a cambios en los enlaces que pudieran alterar el funcionamiento correcto de la misma.

Para los experimentos en general vamos a conectar físicamente los nodos a un subconjunto de *relays* (dado que la arquitectura responde a la misma organización), mientras que los *relays* los conectamos entre ellos. En algunos casos formamos cliques de *relays*, mientras que en otros casos elegimos una cantidad máxima de saltos para la comunicación entre los mismos y creamos una cantidad más acotada de enlaces.

Cada uno de estos enlaces físicos implica una nueva interfaz física en el nodo final, lo cual es importante de tener en cuenta no solo para la ejecución del cliente, sino para entender los resultados y la correspondencia con el mundo real: si formamos una clique de *relays*, estaríamos considerando una red donde cada uno de ellos tiene una interfaz de red dedicada a cada uno de los demás, lo cual no es un caso realista. Sin embargo, dependiendo del experimento, habrán casos donde esta suposición no afecte el objetivo del mismo.

Además, `SherlockFog` nos provee la funcionalidad de poder definir y correr estos nodos en múltiples servidores. Para experimentos con gran cantidad de nodos esto es fundamental: no queremos toparnos con cuellos de botella al ejecutar en un solo servidor. Esto nos permite escalar la red experimental hasta el límite físico de todos los servidores que tengamos disponibles. Al momento de definir el nodo le podemos indicar en qué servidor se debe crear, y debemos también generar un listado de todas las direcciones

IP que deben tomar (que se corresponden con las direcciones de los servidores donde se encuentran).

Con todo esto, ya contamos con lo suficiente para correr con un comando una instancia de la red de `Algorand` de manera controlada, pudiendo distribuirlos sobre distintos servidores, abstrayendo la ejecución de la estructura física de la red. De esta manera podemos correrlo localmente o en clusters que se encuentren interconectados. Alcanza con crear una plantilla de `SherlockFog` para cada uno de los experimentos que queramos correr, y teniendo los nodos creados podemos correr la misma configuración todas las veces que necesitemos.

Por ejemplo, para cada experimento que creamos se adjuntó en este paquete un *script* que se encarga de correr la instancia de `SherlockFog` que lo reproduce. En ellos, se ejecuta un comando similar al siguiente:

```
python3 ../sherlockfog/sherlockfog.py 10nodes_10relays.fog \
# Habilitar modo debug
--debug \
# Utilizar geneve-tunnels para las conexiones entre nodos
--geneve-tunnels \
# Utilizar una interfaz de administraci\'on central
--use-adm-ns \
# Direcci\'on IP de la interfaz de administraci\'on central
--adm-iface-addr 192.168.0.17 \
# Listado de IPs para asignar a los nodos
--real-host-list 10nodes_10relays_ips.txt \
# Utilizar docker para el sistema
--docker \
# Nombre de la imagen docker a utilizar
--docker-image algonodesf:latest \
# Directorio a inyectar en nodos
--docker-storage-bind /networks/10nodes-10relays:/nodes
```

El último parámetro es el que nos permite inyectar el directorio con todas las configuraciones de la red del experimento y que los nodos lean de ella la configuración del cliente que deben utilizar. En ese mismo directorio es donde se guardan los *logs* de la tesis que generan los nodos (en la carpeta `/logs` de ese directorio).

Adicionalmente, podemos definir comandos a correr en cada uno de los nodos o en el servidor que los administra. Por ejemplo, podemos ejecutar *scripts* para aguardar a cierta ronda del sistema para producir un evento en un enlace, o invocar un *script* para inyectar transacciones previamente creadas.

2.7. Poniendo en marcha el experimento

Dependiendo del tamaño del experimento a correr, contamos con dos opciones para hacerlo: en experimentos pequeños podemos utilizar un ambiente local con buenas prestaciones, mientras que para experimentos más grandes, o con mayor demanda de recursos, vamos a utilizar un conjunto de servidores disponibles a través de la red de CloudLab.

A modo de comentario, durante este trabajo también se utilizó la máquina TUPAC, perteneciente al CSC-CONICET, para experimentos de escala mediana. Sin embargo, dada

la naturaleza de acceso a memoria de los clientes, para experimentos grandes debimos recurrir a CloudLab para poder correrlos bajo condiciones normales de la red (la cantidad de cores de cada servidor nos limita la cantidad de nodos que pueden correr paralelamente sin tener que pelear por el acceso a ella).

Controlamos la ejecución de cada experimento conectándonos a alguno de los nodos y utilizando la aplicación `carpenter` provista por `Algorand`, donde visualizamos el estado de cada ronda y los tiempos de generación de cada bloque. Así, podemos verificar que el sistema esté corriendo correctamente y con los tiempos esperados.

2.7.1. Ejecución local

Para la ejecución de experimentos pequeños utilizamos un equipo local que cuenta con las siguientes características:

- Procesador: 16-core Intel i9-12900KF at 3,20 GHz (5,20 GHz max)
- RAM: 64 GB de memoria
- Disco: 1 TB M.2 NVME SSD

Con este equipo podemos realizar simulaciones hasta un tamaño mediano, requiriendo luego de mayor cantidad de recursos para poder emular una red mayor. En lo que pudimos observar de requisitos, los clientes no requieren un uso intensivo de la RAM, pero no queremos que haya cuellos de botella en el acceso a la misma: en cada creación de bloque se produce un acceso simultáneo de todos los nodos a su porción de RAM, pudiendo ser un motivo de degradación en el caso de los tiempos de generación de los mismos.

2.7.2. Ejecución en CloudLab

Si bien los experimentos pequeños (hasta 80 nodos) podemos correrlos localmente, luego necesitamos más recursos para poder mantener la red estable con una generación de bloques que siga la línea de lo visto en el funcionamiento normal. Con este fin, utilizamos clusters de CloudLab para poder ejecutar redes grandes sin problemas relacionados al desempeño.

En esta herramienta podemos crear plantillas con topologías de servidores que nos provee la plataforma durante la duración de un experimento. Para todos los experimentos tratamos de mantener una relación de 20 nodos corriendo por servidor para evitar la saturación del acceso a los recursos del mismo, utilizando servidores con 32 cores y desde 128 GB de memoria. Por ejemplo, para las redes de 150 nodos/50 *relays* y de 300 nodos/100 *relays* utilizamos generalmente 10 y 20 servidores, respectivamente, con las siguientes características:

d6515

- Procesador: 32-core AMD 7452 a 2,35 GHz
- RAM: 128 GB ECC Memory (8×16 GB 3200MT/s RDIMMs)
- Disco: Two 480 GB 6G SATA SSD
- NIC: Dual-port Mellanox ConnectX-5 100 Gbit NIC (PCIe v4.0)
- NIC: Dual-port Broadcom 57414 25 Gbit NIC

CloudLab cuenta con 28 servidores físicos de estas características que pueden ser reservados por una cantidad predefinida de horas, lo cual nos permite planear nuestros experimentos con anticipación y disponer de los recursos para correrlos.

En su sitio contamos con la posibilidad de crear plantillas de topologías físicas para ciertas cantidades de servidores. Las escribimos en XML, definiendo cada nodo con su nombre, su tipo de *hardware* (máquinas d6515), sus interfaces de red, si debe contar con una IP routeable a través de internet (de manera que tenga una IP pública). Todos los nodos se conectan con enlaces Ethernet a la misma red LAN, donde habilitamos *jumbo-frames*, opción que habilita el uso de frames de gran tamaño (hasta 9000 B MTU).

Con cada asignación, los servidores reinician su contenido, ofreciendo una imagen Ubuntu limpia lista para ser utilizada. Por este motivo debimos desarrollar una serie de *script*s que instalen todas las dependencias necesarias para correr `SherlockFog`, configuren los accesos SSH entre los nodos, descarguen las configuraciones de las redes experimentales y todos los *script*s que las corren. También, sumamos *script*s para poder realizar la descarga de los resultados de cada uno de los servidores, aprovechando la conexión a internet de cada uno de ellos y evitando centralizar todos en el mismo y así solo utilizar la velocidad de subida de uno solo.

En una ejecución normal, nos conectamos por SSH a uno de los servidores, transferimos un paquete que contiene todo lo descrito previamente, y desde él corremos el *script* de instalación de los servidores. Una vez que finaliza, ya podemos correr los experimentos, distribuyendo la carga sobre todos los servidores físicos sin que la red lógica esté al tanto gracias a `SherlockFog`.

2.8. Validación

Para asegurarnos de que la metodología elegida cumpla con todo lo que planteamos, elaboramos un conjunto de validaciones que nos permitirán, para cada experimento de las secciones *Estabilidad de la red ante cambios en la topología física* y *Reconstruyendo la topología lógica*, verificar el funcionamiento esperado.

Los puntos a validar serán los siguientes:

1. El servidor crea un contenedor `docker` para cada nodo.
2. La red física se crea correctamente según el *script* de `SherlockFog`.
3. Los nodos administran cada uno la cuenta que se le asignó en la creación de la red.
4. Los nodos mantienen la cantidad de *stake* asignada al principio del experimento.
5. Los nodos se comunican sólo con el *relay* que tienen asignado.
6. Capturamos correctamente el instante en el que se genera cada bloque en la red con nuestro código.
7. La red lógica de `Algorand` avanza y mantiene un tiempo de bloque estable y cercano al tiempo real (alrededor de 4,36 s sin transacciones o con una baja cantidad de ellas).
8. Todos los nodos de la red participan del protocolo de consenso.

9. Las transacciones inyectadas son reconocidas por el sistema y son agregadas a un bloque.

10. Los nodos guardan el *log* de los eventos activados en el archivo de *logs* personalizado.

11. Podemos replicar el mismo funcionamiento en un conjunto de servidores de CloudLab.

12. `SherlockFog` es capaz de aplicar demoras en los enlaces y simular cortes en los mismos con demoras grandes.

13. Nuestro módulo en el cliente de `Algorand` captura todos los mensajes recibidos de los demás nodos.

14. Los nodos de `Algorand` se conectan a cuatro *relays* por defecto en toda su ejecución, sin cambiar por otros.

Algunos de estos puntos responden al funcionamiento de nuestros *script*s para correr la red correctamente, mientras que otros hacen referencia a las herramientas que utilizamos para alterar la red y capturar los eventos de la misma. Por otro lado, el punto trece hace referencia a una característica de la red que asumimos como cierta para nuestros experimentos: los nodos en `Algorand` durante su ejecución se mantienen conectados al mismo conjunto de *relays* sin cambiarlos excepto que detecten alguna falla en ellos o mejoras de desempeño en otro disponible.

Para los primeros once, diseñamos un experimento donde corremos una red de diez nodos y diez *relays*, dejándola correr durante 30 min. En el transcurso de la misma inyectamos transacciones para verificar que sean recibidas por los nodos y se guarden en bloques posteriormente.

Organizamos la topología física para que los *relays* estén conectados en forma de anillo, mientras que los nodos se conectan a un *relay* cada uno físicamente, pero a cuatro de manera lógica. Distribuimos el *stake* de manera homogénea entre las cuentas.

Elegimos enviar montos muy pequeños de *algos* para verificar las transacciones de manera de mantener lo más posible la distribución equitativa del *stake* en la red en toda la duración del experimento, pero dándonos lugar para revisar el comportamiento durante su ejecución (enviamos 100 `algo`, que representan el 0,000 000 000 000 1 % de su *stake*).

Sistema:

- **Nodos:** 10 (n1 a n10)
- ***Relays*:** 10 (n11 a n20)
- ***Relays* conectados por nodo:** 1 lógico, 1 físico (20 ms)
- ***Relays* conectados por *relay*:** 4 lógicos, 1 físico (30 ms)
- ***Stake*:** homogéneo
- **Tiempo de ejecución:** 30 min
- **Servidor:** local (1 servidor)
- **Transacciones inyectadas:**
  - Ronda 50: 100 *algos* de n1 a n2.
  - Ronda 100: 100 *algos* de n2 a n1.

Utilizamos las herramientas provistas por `Algorand` para hacer validaciones durante el experimento. Finalizado el mismo, utilizamos los *logs* para poder analizar las estadísticas globales.

Para los demás puntos, aclaramos los sistemas utilizados en cada una de las descripciones. Veamos entonces ítem por ítem los temas a validar.

2.8.1. *"El servidor crea un contenedor `docker` para cada nodo"*

Esperamos: ver que se creó un contenedor para cada uno de los nodos de la red que estamos emulando, del n1 al n20.

Corremos: `sudo docker container list`

Verificamos:

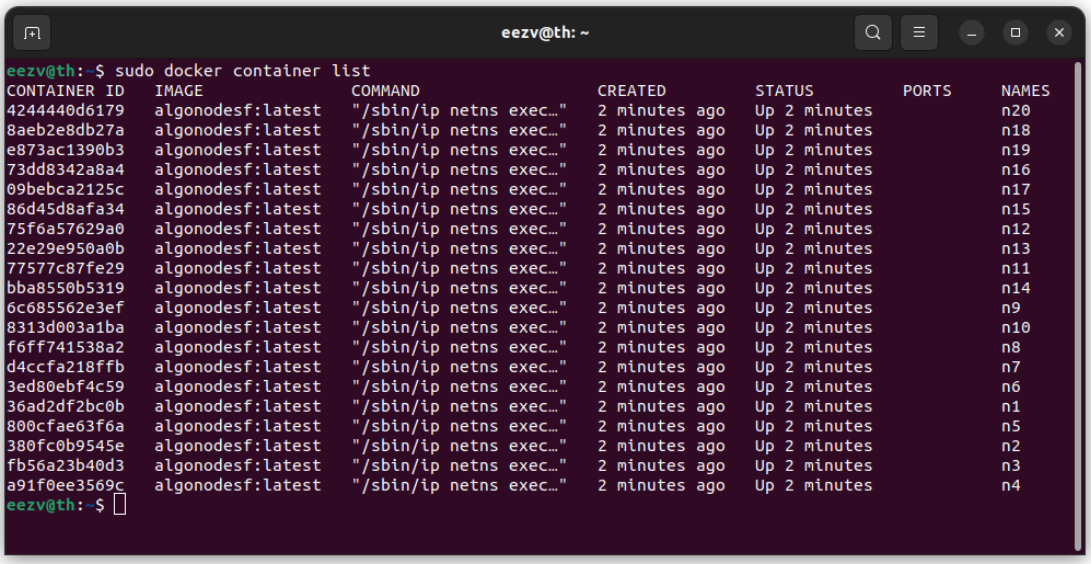


Fig. 2.2: Listado de contenedores `docker` ejecutando en el servidor local

Vemos en la figura 2.2 que efectivamente `SherlockFog` crea un contenedor `docker` por cada uno de los nodos que emulamos, viendo que en la columna "NAMES" aparecen todos los nodos del n1 al n20, ejecutando la imagen de *algonodesf* que creamos inicialmente.

2.8.2. *"La red física se crea correctamente según el* script *de `SherlockFog`"*

Esperamos: que la topología física se cree con un anillo para los nodos n11 a n20, mientras que los nodos n1 a n10 se conectan a un solo *relay*.

Corremos: agregamos el comando `savegraph 10n_10r_circle_validation.svg` en el *script* de `SherlockFog`, que grafica la topología física generada por la herramienta.

Verificamos:

Luego de creada la topología física, `SherlockFog` crea el siguiente gráfico para la misma:

Comprobamos que los nodos n11 a n20 se encuentran conectados con un solo enlace en forma de anillo según la imagen 2.3, mientras que los nodos n1 a n10 se conectan directamente a uno de los *relays*.

2.8.3. *"Los nodos administran cada uno la cuenta que se le asignó en la creación de la red"*

Esperamos: que cada nodo maneje solamente una cuenta y que sea la que asignamos en la creación de la configuración de red del sistema. Por ejemplo, el nodo n1 debe correr la cuenta WQO7KXAKCH7GTP6JZLU3QLTQPGCQSWFDKMNKH4Z2RX2NR3X3CW33WF564E, mientras que el nodo n2 debe correr UOBQ6Z4ODDATQAHTVNLJGU5EW3RJVWY6R2X7H3U5563CQQZPD7ZCDV3JW4.

Corremos: `goal account list` en los nodos n1 y n2, que lista las cuentas administradas por un cliente.

Verificamos:

En ambos casos, para las salidas de las figuras 2.4 y 2.5, encontramos una sola cuenta administrada por cada nodo, que coincide con la que les fue configurada en la creación de la red.

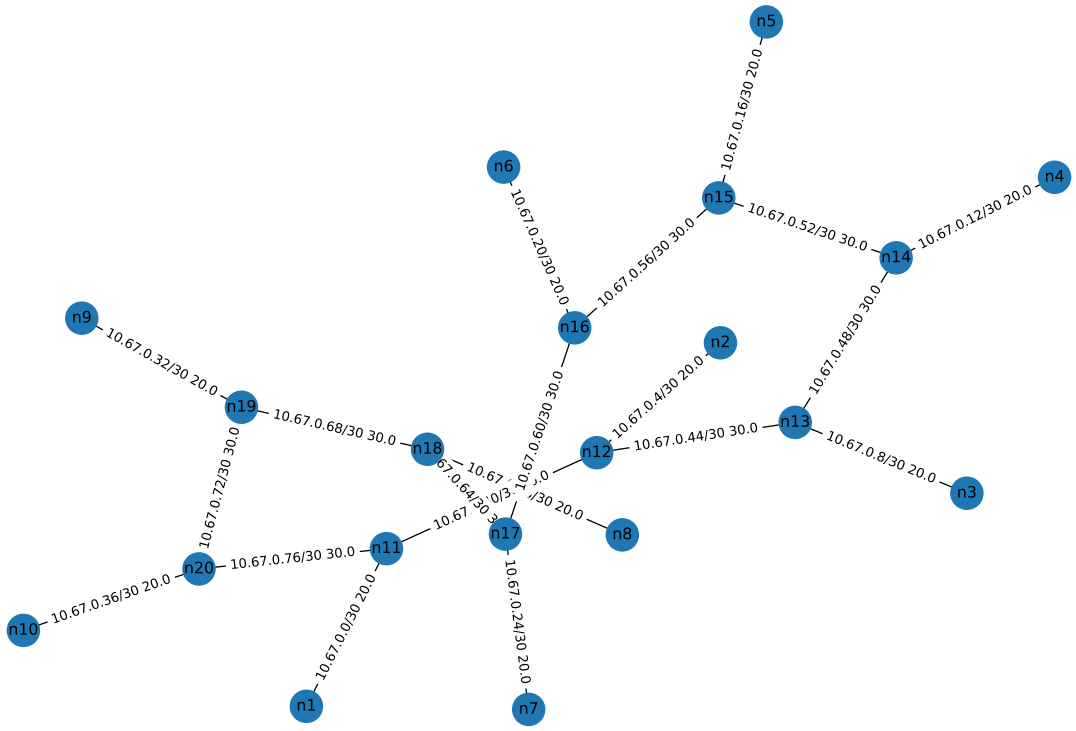


Fig. 2.3: Topología física emulada por `SherlockFog` para el sistema

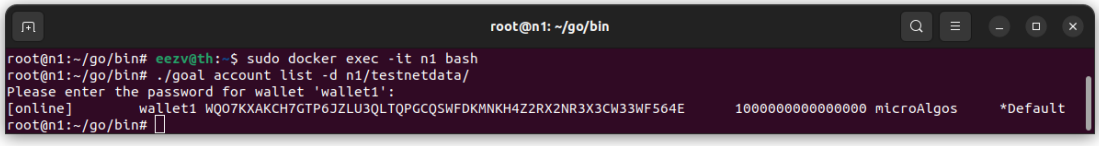


Fig. 2.4: Listado de cuentas administradas por el nodo n1

2.8.4. *"Los nodos mantienen la cantidad de stake asignada al principio del experimento"*

Esperamos: que cada nodo maneje no solo la cuenta asignada, sino también el *stake* que se le asignó a la misma (los *relays* tienen cero bajo su control).

Corremos: `goal account list --info` en los nodos n1 y n2, y uno de los *relays* (n12), que lista las cuentas administradas por un cliente con el detalle de los montos que contiene la misma.

Verificamos:

En el caso de los nodos n1 y n2, representados en las figuras 2.6 y 2.7, verificamos que participan de la red con la misma cantidad de *algos* en su poder (1000000000, o 1000000000000000 *microAlgos*). Aparte de ese monto, no contienen otros *assets*.

En cambio, el *relay* n12 no maneja *stake* ni de ningún otro activo en la red, observado en la imagen 2.8.

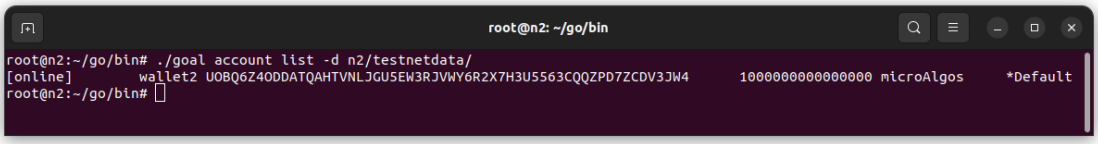


Fig. 2.5: Listado de cuentas administradas por el nodo n2

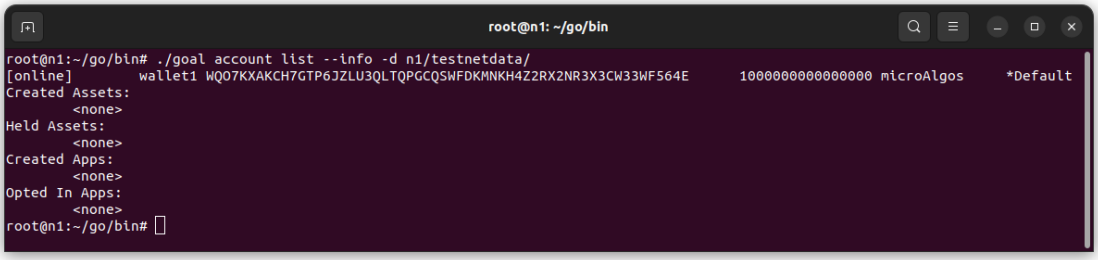


Fig. 2.6: Listado de cuentas administradas por el nodo n1, junto con su *stake* y otros *assets*

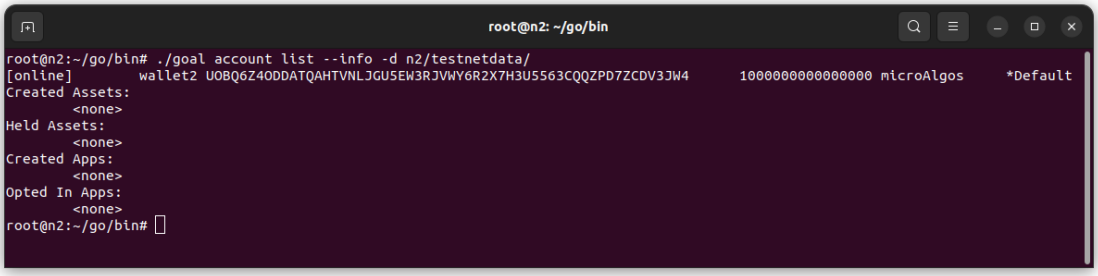


Fig. 2.7: Listado de cuentas administradas por el nodo n2, junto con su *stake* y otros *assets*

2.8.5. *"Los nodos se comunican sólo con el relay que tienen asignado"*

Esperamos: que cada nodo se conecte solamente con el *relay* que le fue asignado. En el caso del n1 debe conectarse al *relay* n11, mientras que el n2 al n12.

Corremos: `netstat -tup` sobre dos de los nodos, debiendo aparecer solamente la IP del *relay* conectado.

Verificamos:

Vemos en las figuras 2.9 y 2.10 que ambos nodos se comunican directamente con el único *relay* que tienen configurado como preveíamos, manteniéndolo durante toda la duración del experimento.

2.8.6. *"Capturamos correctamente el instante en el que se genera cada bloque en la red con nuestro código"*

Esperamos: que nuestro código dentro del cliente de `Algorand` registre correctamente el momento en el que se genera cada bloque en la red, además de registrar quién fue el

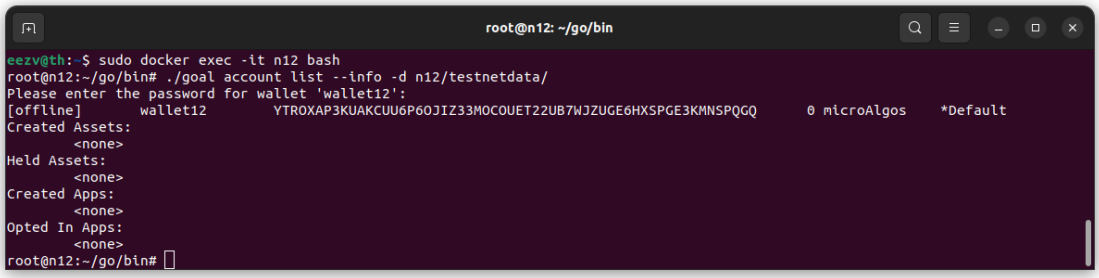


Fig. 2.8: Listado de cuentas administradas por el *relay* n12, junto con su *stake* y otros *assets*

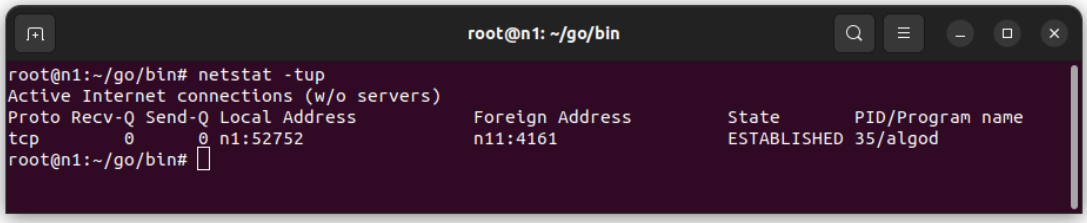


Fig. 2.9: Conexiones activas del nodo n1 durante la ejecución del experimento.

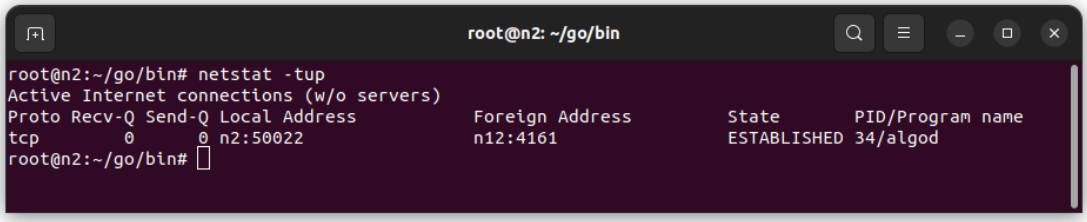


Fig. 2.10: Conexiones activas del nodo n2 durante la ejecución del experimento.

que lo propuso para poder analizar la distribución de los mismos.

Corremos: el experimento planteado y comparamos los *logs* generados por defecto en el *relay* n11 y los generados por nuestro código, tomando diez ejemplos de rondas distribuidas en el tiempo de ejecución como muestras del mismo.

Verificamos:

Para analizar los registros, tomamos una muestra de diez bloques para comparar los tiempos registrados en cada caso. Inicialmente comparamos ambos *logs*, encontrando que ambos coinciden en la cantidad total de bloques guardados: 377.

Del log de `Algorand`:

```
{
    "Context":"Agreement",
    "Hash":"OQSPE7EIAD2CJDESWSLRZVQ2T3RQAEMQ2YFVBBDIB33L5EU5SUYA",
    ...
    "Round":377,
    "Sender":"GJPINLGOOPINRCNQFOOFCB5WVFMZEKM3RGPDX4ARDUSMILHNVH6C2ODZCU",
    ...
    "msg":"finished round 377",
    "time":"2023-08-06T19:49:43.083306Z"
}
```

De nuestro log:

```
{
    ...
    "msg":"9-ENSURING_BLOCK-377-
    OQSPE7EIAD2CJDESWSLRZVQ2T3RQAEMQ2YFVBBDIB33L5EU5SUYA-
    GJPINLGOOPINRCNQFOOFCB5WVFMZEKM3RGPDX4ARDUSMILHNVH6C2ODZCU-
    none",
    "time":"2023-08-06T19:49:43.083220Z"
}
```

En nuestro *log* guardamos el tipo de evento (*ENSURING_BLOCK*), el número de ronda (377), el hash del bloque (`OQSPE7...U5SUYA`), la cuenta que lo propuso (`GJPINL...2ODZCU`), las transacciones que pudiera contener (en este caso se indica *none* al no poseer ninguna) y finalmente el momento en el que fue creado. Vemos que coincide plenamente con lo registrado en el *log* de `Algorand`, habiendo una mínima diferencia en el tiempo debido a que nuestro *log* se ejecuta antes que el del cliente, pero que resulta despreciable.

Comparamos los tiempos de generación y la cuenta que lo generó para las rondas 5, 42, 79, 116, 153, 190, 227, 264, 301 y 338, tanto para los *logs* de `Algorand` como los personalizados para este trabajo de tesis.

En todos los casos, el emisor del bloque coincide para ambos *logs*, al igual que el tiempo de generación (nuevamente con una mínima diferencia debido a la ubicación del código que realiza cada acción). En base a estos resultados, podemos confiar en que nuestro código reconoce correctamente el momento de creación de los bloques junto con su emisor.

Ronda	Emisor `Algorand`	Tiempo `Algorand`	Emisor tesis	Tiempo tesis
5	UOBQ6Z...DV3JW4	19:22:51.574669	UOBQ6Z...DV3JW4	19:22:51.574599
42	UOBQ6Z...DV3JW4	19:25:31.873160	UOBQ6Z...DV3JW4	19:25:31.873058
79	LES37P...J6PHUU	19:28:12.187157	LES37P...J6PHUU	19:28:12.187032
116	VDTZEZ...WEKTTI	19:30:52.432804	VDTZEZ...WEKTTI	19:30:52.432728
153	UOBQ6Z...DV3JW4	19:33:32.652216	UOBQ6Z...DV3JW4	19:33:32.652129
190	LES37P...J6PHUU	19:36:12.944778	LES37P...J6PHUU	19:36:12.944593
227	GSEPOG...7V5MEY	19:38:53.196143	GSEPOG...7V5MEY	19:38:53.196067
264	MTI4DF...OKIDCM	19:41:33.537997	MTI4DF...OKIDCM	19:41:33.537928
301	MTI4DF...OKIDCM	19:44:13.884305	MTI4DF...OKIDCM	19:44:13.884194
338	WQO7KX...WF564E	19:46:54.136353	WQO7KX...WF564E	19:46:54.136272

Tab. 2.1: Emisor y tiempo de emisión de los bloques para las rondas dadas, tanto según los *logs* de `Algorand` como de nuestro código para el trabajo de tesis, coincidiendo en ambos valores con una diferencia despreciable en el tiempo.

2.8.7. *"La red lógica de `Algorand` avanza y mantiene un tiempo de bloque estable y cercano al tiempo real (alrededor de* 4,36 s *sin transacciones o con una baja cantidad de ellas)"*

Esperamos: que la red avance generando bloques nuevos cada aproximadamente 4,36 s.

Corremos: `carpenter` que muestra en tiempo real los *logs* generados por el cliente de `Algorand`, con el estado de cada ronda y una marca temporal. Finalizado el experimento, analizamos los *logs* con nuestras herramientas para conocer el tiempo de generación de cada bloque.

Verificamos: Tomamos dos momentos del experimento para ver el tiempo de ronda: cerca del principio del mismo, y unos 20 min más tarde.

Para los dos momentos de las figuras 2.11 y 2.12 el tiempo de ronda resulta casi idéntico, siendo de 4,3 s.

Finalizada la experiencia, tomamos los registros de todos los tiempos de ronda para analizar si hubieron tiempos que se desviaran de ese valor. Vemos que el tiempo de ronda resulta muy consistente, según la salida mostrada en 2.13, tomando desde apenas menos que 4,28 s hasta un poco más de 4,34 s. Estos valores se encuentran muy cercanos a la marca de 4,36 s que buscábamos en el funcionamiento de nuestra red.

A pesar de lograr un resultado optimista, nos interesa entender por qué nuestro tiempo de ronda resulta consistentemente menor que el promedio de 4,5 s según los datos provistos por la comunidad. En base a documentación, vemos que el tiempo de ronda depende fuertemente de la cantidad de transacciones e información que se agregue en el bloque propuesto: cada transacción debe ser verificada y analizada, lo cual lleva un cierto tiempo. En nuestros ejemplos todos los bloques se encuentran vacíos y además hemos quitado del código la validación de transacciones para evitar sobrecargar el sistema con chequeos innecesarios.

Por eso, planteamos un nuevo experimento donde restauramos las validaciones e inyectamos una gran cantidad de transacciones por bloque para ver si esto tiene impacto en el tiempo de ronda (y así justificar la diferencia entre nuestros tiempos previos y los buscados). Utilizamos un sistema de 30 nodos y 10 *relays*, donde cada nodo se conecta a cuatro *relays* y los *relays* se conectan a cuatro de sus pares, y en durante el mismo,

entre las rondas 200 y 400, inyectamos 2500 transacciones por bloque (entendiendo que la cantidad máxima es de 5000 por bloque según la documentación).

Sistema:

- **Nodos:** 30
- ***Relays*:** 10
- ***Relays* conectados por nodo:** 4 lógicos, 4 físicos
- ***Relays* conectados por *relay*:** 4 lógicos, 4 físicos
- ***Stake*:** homogéneo

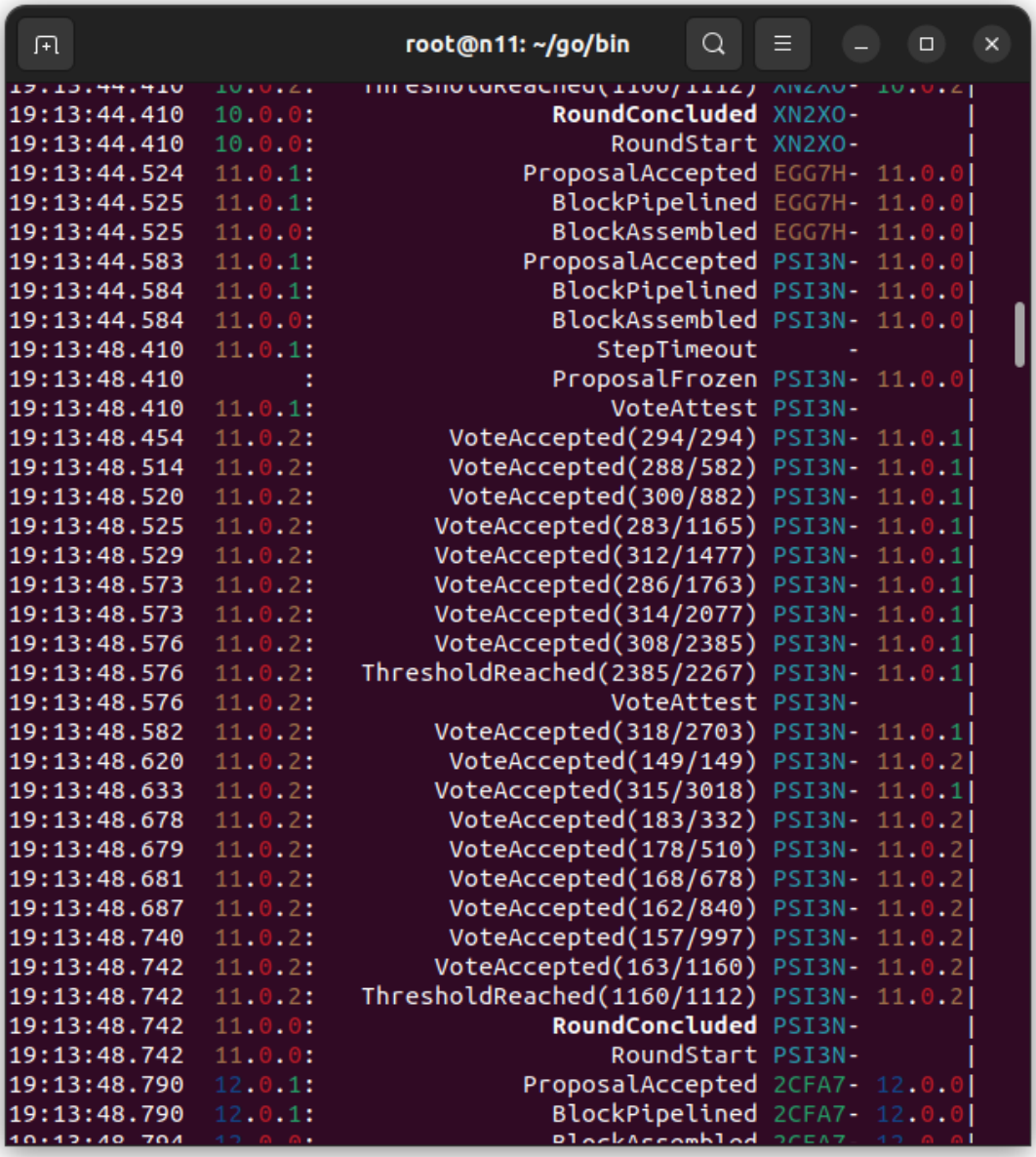


Fig. 2.11: Logs observados con `carpenter` en el comienzo del experimento, donde se muestra el inicio y fin de la ronda 11, durando alrededor de 4,3 s.

- **Tiempo de ejecución:** 600 rondas (45 min)
- **Transacciones por ronda (tpr):**
  - Ronda 200 a 400: 2500 tpr.
- **Servidor:** local (1 servidor)

Vemos en la figura 2.14 el impacto en el tiempo de ronda que tiene la inyección de transacciones: antes de la ronda 200, el tiempo promedio es de 4,3 s, pero al comenzar a incorporar más información en los bloques ese tiempo asciende a 4,7 s (con algunos valores altos que pueden deberse a la gran carga inicial de transacciones). Este valor

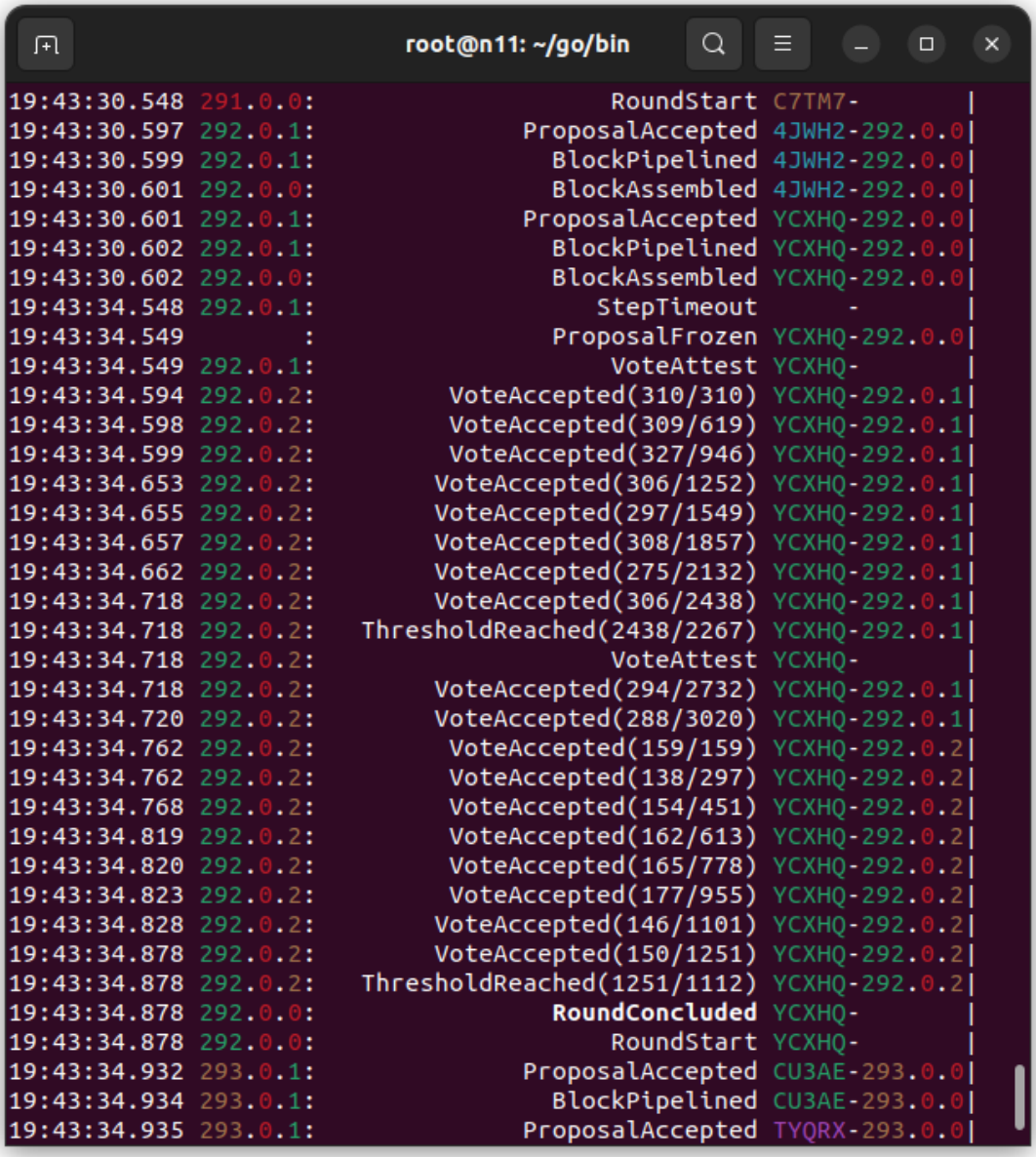


Fig. 2.12: Logs observados con `carpenter` más adelante en el experimento, donde se muestra el inicio y fin de la ronda 292, durando alrededor de 4,3 s.

resulta superior al de 4,5 s que estima la documentación, pero que nos ayuda a entender que existe una dependencia clara entre la cantidad de transacciones y el tiempo de ronda.

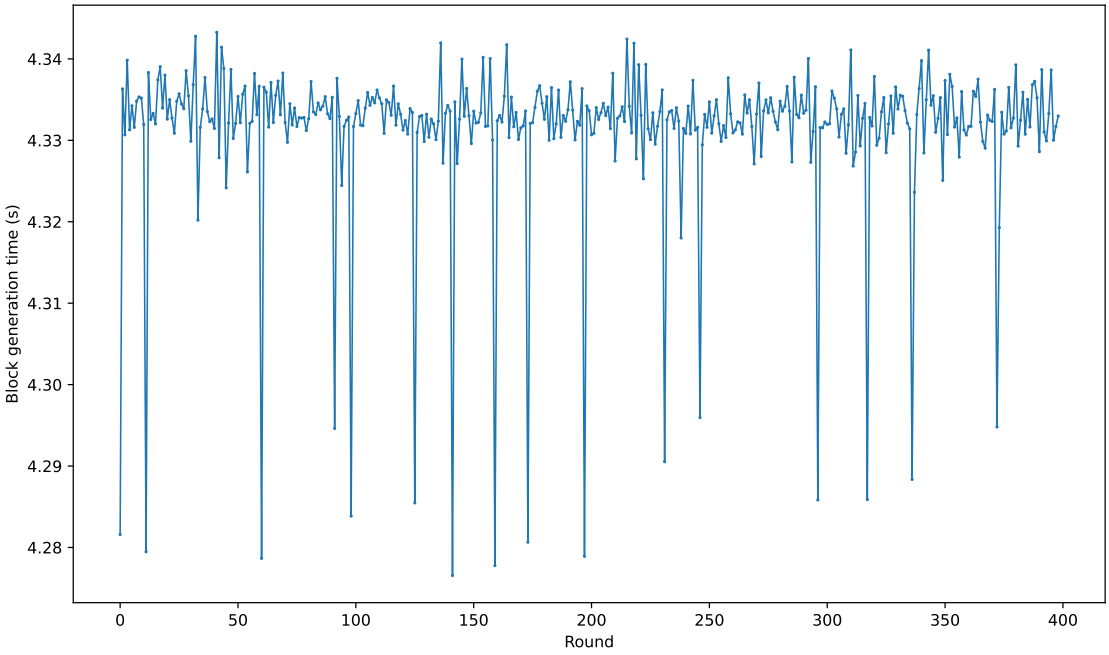


Fig. 2.13: Tiempo de generación para cada bloque del experimento.

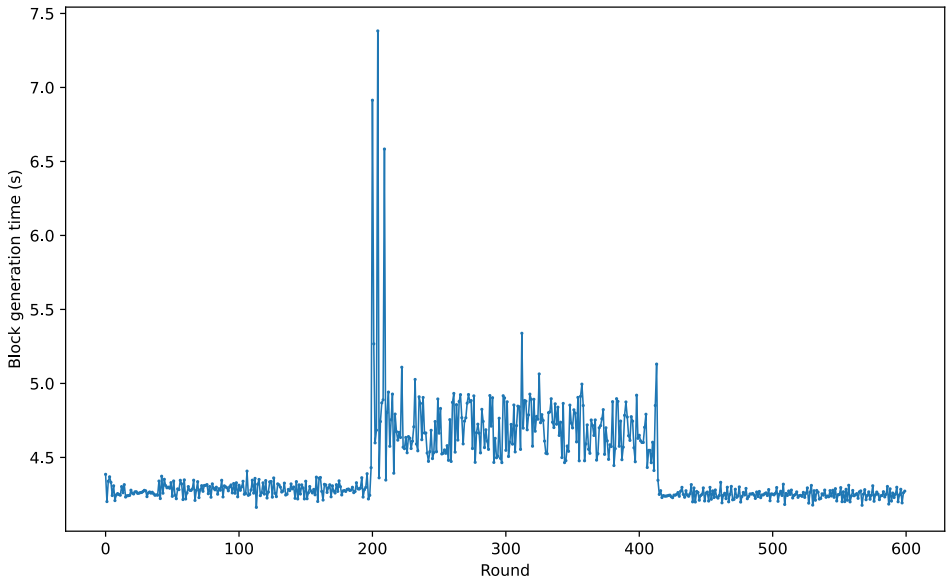


Fig. 2.14: Tiempo de generación para cada bloque del experimento con inyección de transacciones. A partir de la ronda 200 se inyectan 2500 transacciones por ronda, aumentando el tiempo de generación de bloque de 4,3 s a 4,7 s en promedio.

Por este motivo, la diferencia que encontramos puede deberse a una estimación de menor cantidad de información por bloque en promedio para el estado de la red en ese entonces. Luego, si consideramos ambos tiempos de ronda, si los bloques tuvieran entre cero y 2500 transacciones en promedio, el valor de 4,5 s cobra sentido. Finalmente, al detener la inyección, el tiempo por ronda desciende al inicial.

Con esta experiencia podemos confirmar que nuestro tiempo, a pesar de ser menor que el promedio para la red real, coincide con lo esperado para bloques vacíos o pocas transacciones. Así, podemos confiar en que el funcionamiento de la red es correcto para este valor.

2.8.8. *"Todos los nodos de la red participan del protocolo de consenso de manera equitativa"*

Esperamos: que todos los nodos hayan propuesto una cantidad similar de bloques durante el experimento, debido a la distribución equitativa del *stake*.

Corremos: el *script* de análisis de bloques sobre los *logs* generados en el experimento una vez que finalizó.

Verificamos:

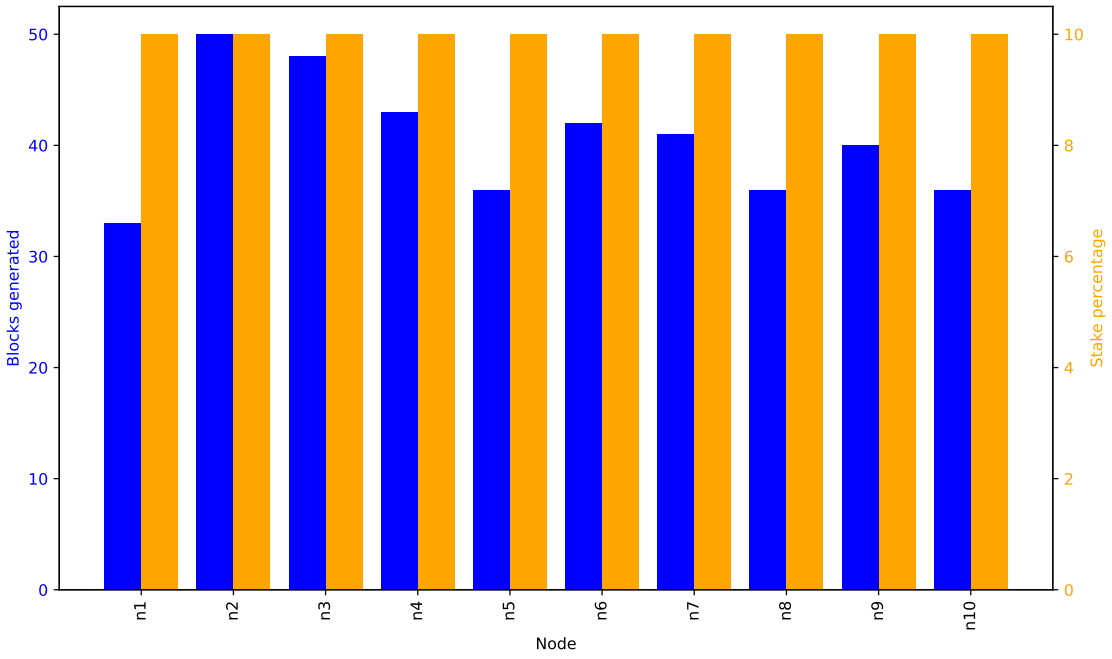


Fig. 2.15: Cantidad de bloques propuestos por nodo, junto a su *stake*.

En la distribución mostrada en el gráfico 2.15, todos los nodos participan del avance de la red y de forma equitativa, teniendo diferencias debido a la característica aleatoria de la selección de los mismos, pero que con la duración del experimento (30 min) no resultan importantes. Se grafica también el *stake* de cada uno de los nodos, confirmando la distribución homogénea del mismo.

2.8.9. *"Las transacciones inyectadas son reconocidas por el sistema y son agregadas a un bloque"*

Esperamos: que al inyectar dos transacciones generadas previamente en el sistema, las mismas son recibidas por el nodo destino y son agregadas a un bloque. Luego, el *stake* de las cuentas utilizadas se modifica.

Corremos: agregamos el envío de transacciones en el *script* de `SherlockFog` para que se inyecten en las rondas 50 y 100. Apenas se ejecuta, corremos `goal node pendingtxns` en el nodo receptor para verificar que la transacción haya sido recibida y se esté procesando. Luego, utilizamos la API expuesta por los nodos para verificar que la transacción haya sido agregada en el bloque (`GET /v2/transactions/pending/{txid}`). Repetimos para la otra transacción en la ronda 100.

Verificamos: Se envía la transacción de 100 `algo` (o 100 000 000 `µalgo`) de la cuenta del nodo n1 a la del n2, para que sea procesada por el nodo n1. Al momento de ser enviada, revisamos las transacciones pendientes en n1:

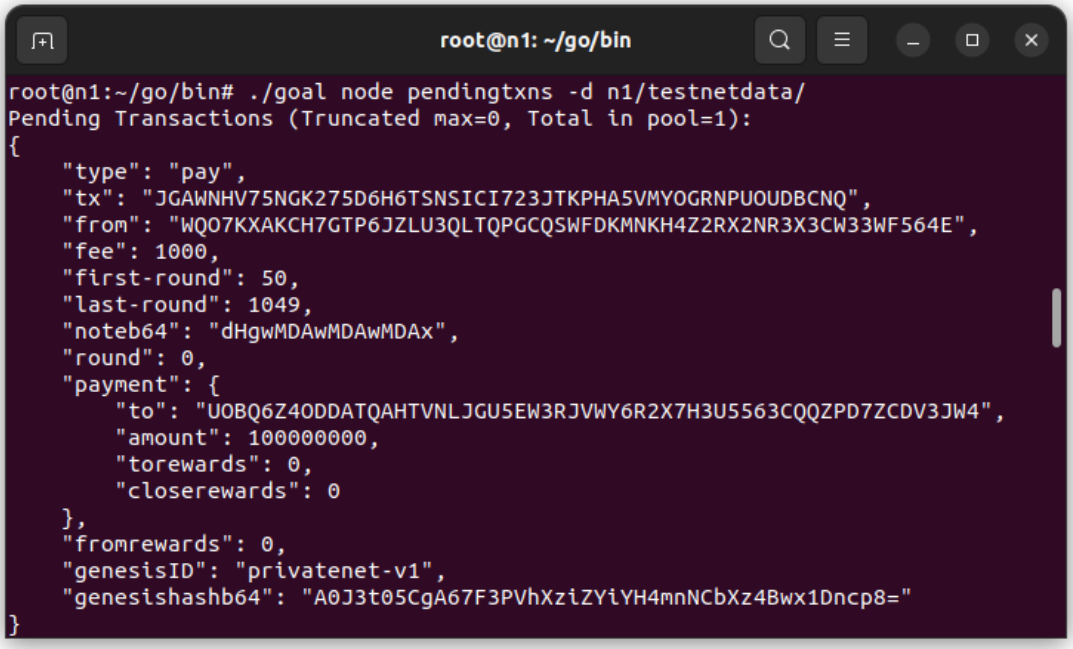


Fig. 2.16: Transacciones pendientes en nodo n1 en ronda 50.

La transacción enviada aparece, en 2.16, como recibida en el nodo, pero pendiente de procesamiento. Aquí tenemos el identificador de la misma (el valor del campo "tx", JGA...BCNQ) que nos sirve para consultar sobre su estado a través de la `API Rest` del cliente `Algorand`.

Si bien la transacción fue enviada al nodo n1 en la ronda 50, la misma termina siendo agregada a un bloque (y así confirmada) en la ronda 52 para la figura 2.17. También se encuentra el detalle de la cuenta emisora y la receptora del intercambio.

Repetimos luego la experiencia en la ronda 100, haciendo una transacción de la cuenta del nodo n2 a la del n1, enviándola al nodo n2 para su procesamiento, viendo en 2.18 y 2.19 la salida de los comandos ejecutados.

El nodo n2 la recibe, procesa y finalmente es confirmada en el bloque de la ronda 102, nuevamente dos bloques luego de ser recibida. Esto debemos tenerlo en cuenta al momento de hacer nuestros experimentos, no es inmediato el procesamiento de las transacciones, se requiere al menos un bloque de tiempo para ser procesada (se recibe en la ronda 100, se procesa en la 101 y es agregada en el 102).

Para que esto sea escalable, no nos alcanza con consultar manualmente por el estado de cada transacción pendiente inmediatamente luego de ser enviada para obtener el identificador de la misma. Por ello, cuando la herramienta de envío de transacciones envía una, escribe en el directorio indicado como salida un archivo que contiene la respuesta obtenida por el nodo (representado por la imagen 2.20). Esta respuesta, mostrada en 2.21, contiene el identificador que se le dio a la transacción cuando fue recibida.

Con este valor podemos consultar a los nodos sobre el estado de la transacción y verificar en qué momento fue confirmada, así como también analizar si pudo haber fallado su envío (en ese caso, no veremos el identificador, sino que tendremos un mensaje de error). Esto podría suceder si el *fee* pagado para su procesamiento fuera muy bajo.

2.8.10. *"Los nodos guardan el log de los eventos activados en el archivo de* logs *personalizado"*

Esperamos: que al finalizar el experimento, todos los nodos hayan guardado sus *logs* y postmortem en la carpeta de configuración de red compartida por todos.

Corremos: revisamos el directorio correspondiente cuando finaliza el experimento.

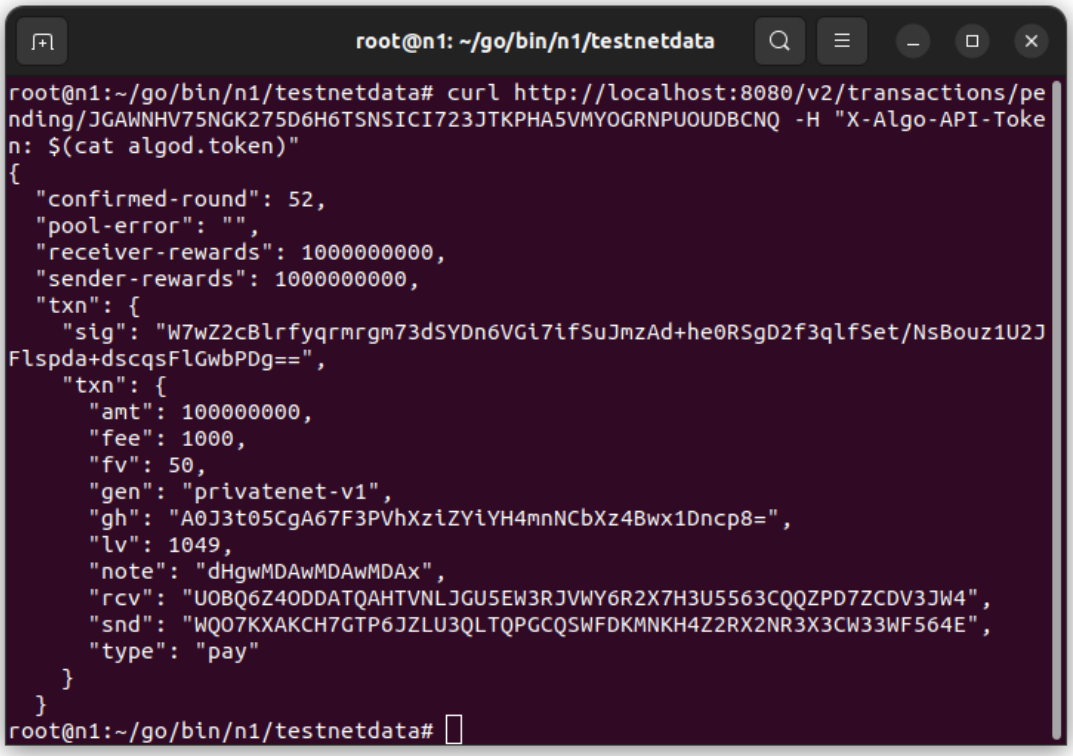


Fig. 2.17: Confirmación de la transacción enviada en la ronda 50, confirmada en la ronda 52.

Verificamos: Finalizado el experimento, revisamos el directorio de *logs* en la carpeta de la configuración de red, encontrando todos los archivos correspondientes a cada uno de los nodos en la figura 2.22.

También verificamos lo mismo para los post-mortem de todos ellos, visto en 2.23.

Esta información nos permite luego analizar los eventos que hayamos seleccionado en el cliente de `Algorand` y también reconstruir cualquier problema que haya podido tener la red a partir de los *logs* propios del sistema.

2.8.11. *"Podemos replicar el mismo funcionamiento en un conjunto de servidores de CloudLab"*

Esperamos: que si distribuimos la red en varios servidores de CloudLab, obtenemos los mismos tiempos de ronda que en el servidor local, y la misma distribución de la participación de cada cuenta.

Corremos: ejecutamos el mismo experimento en tres servidores de CloudLab y luego analizamos los *logs* con nuestras herramientas.

Verificamos: Con `carpenter` vemos los tiempos de ronda durante la ejecución, para las mismas rondas elegidas en el experimento local, en la figuras 2.24 y 2.25.

Los tiempos de ronda en ambos casos coinciden con los que calculamos en el sistema local. Analizamos los tiempos para todo el experimento, resultando en lo visto en la figura 2.26.

Nuevamente, el resultado es similar al experimento local, teniendo tiempos de ronda cercanos a los 4,3 s. Validamos también la participación de los nodos en este avance en la imagen 2.27.

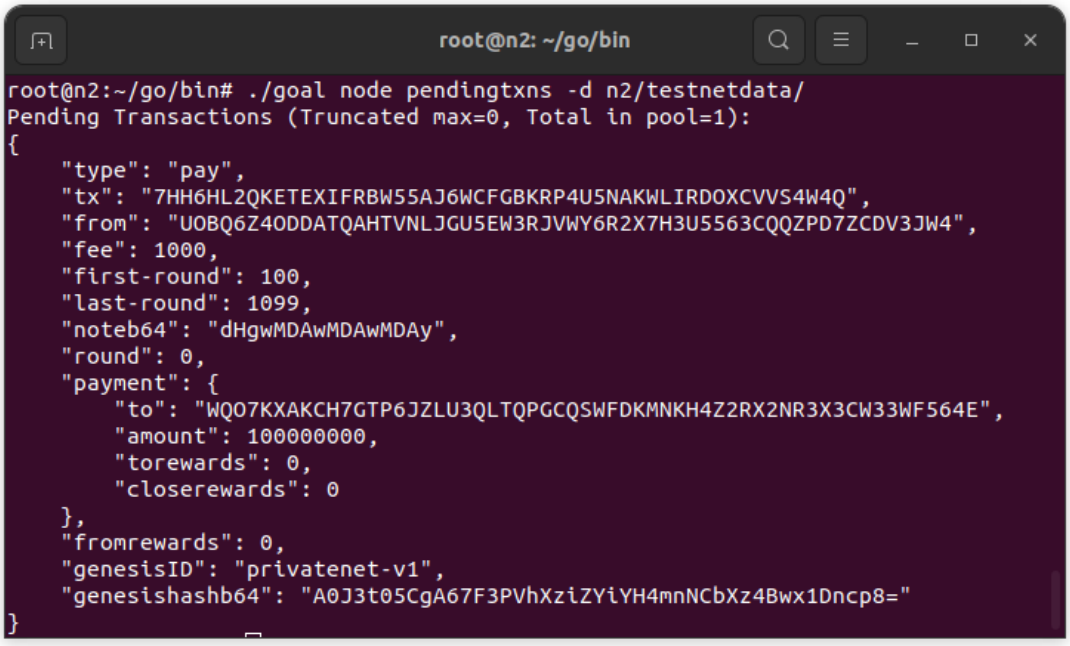


Fig. 2.18: Transacciones pendientes en nodo n2 en ronda 100.

La participación de los nodos resulta equitativa, correspondiente a la distribución homogénea del *stake*.

Con esto confirmamos el correcto funcionamiento de nuestros *script*s y configuraciones

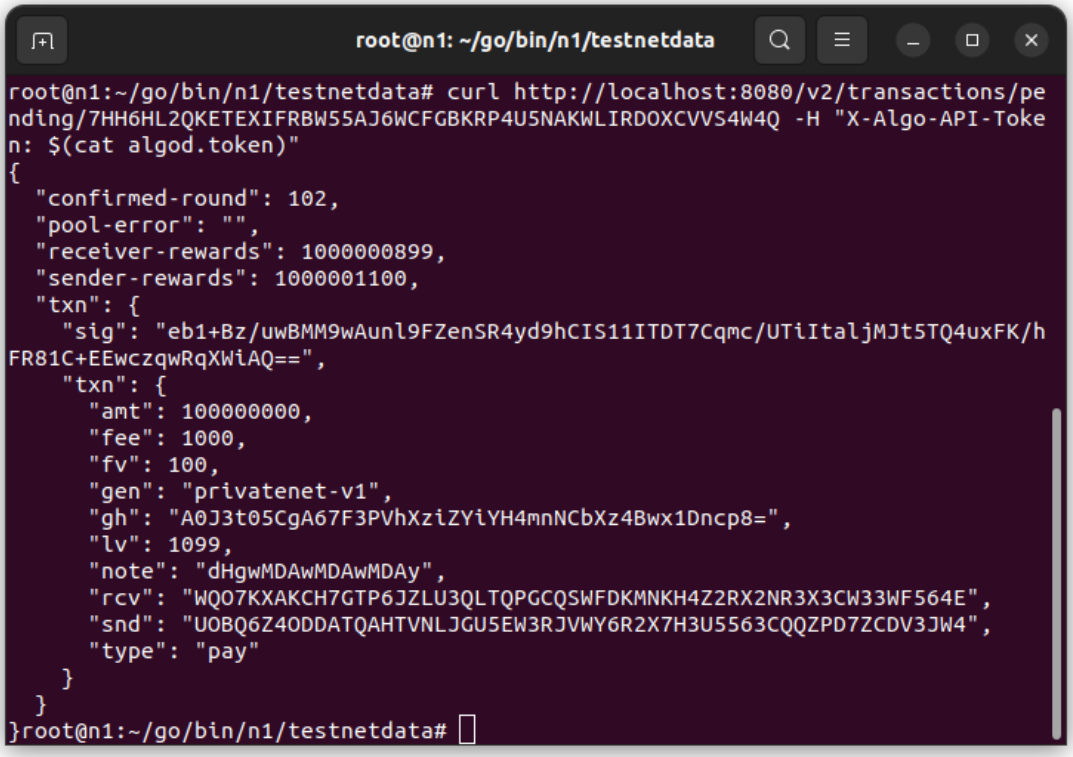


Fig. 2.19: Confirmación de la transacción enviada en la ronda 100, confirmada en la ronda 102.

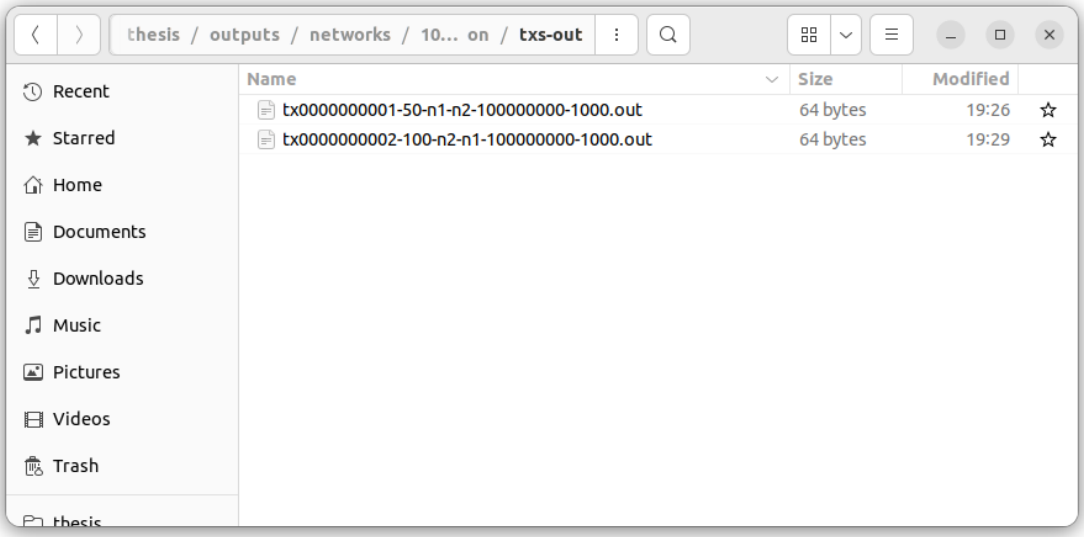


Fig. 2.20: Directorio con la respuesta del envío de cada transacción a la red.

para correr instancias privadas de redes de `Algorand` sobre las topologías emuladas de `SherlockFog`, tanto en ambientes locales, como distribuida en servidores de CloudLab.

2.8.12. *"SherlockFog es capaz de aplicar demoras en los enlaces y simular cortes en los mismos con demoras grandes"*

En la primer parte de este trabajo vamos a experimentar agregando latencias en los enlaces de nuestra red y simulando cortes en los mismos al agregar tiempos de demora

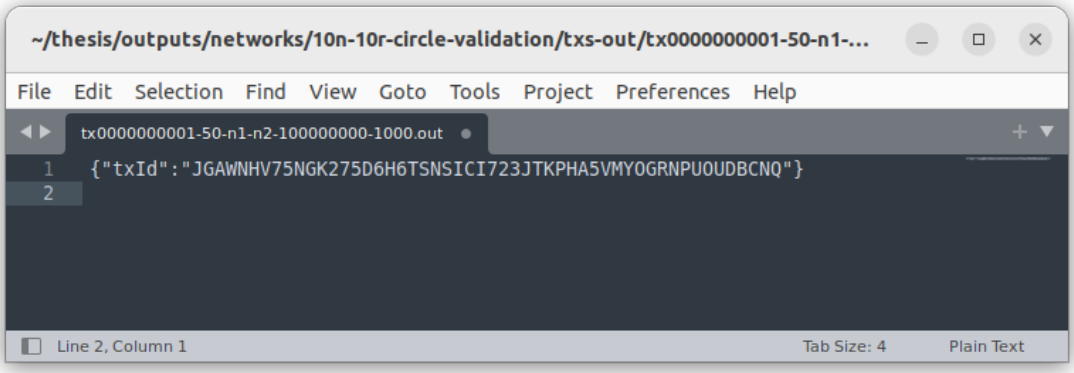


Fig. 2.21: Contenido del archivo de respuesta al envío de la transacción de la ronda 50.

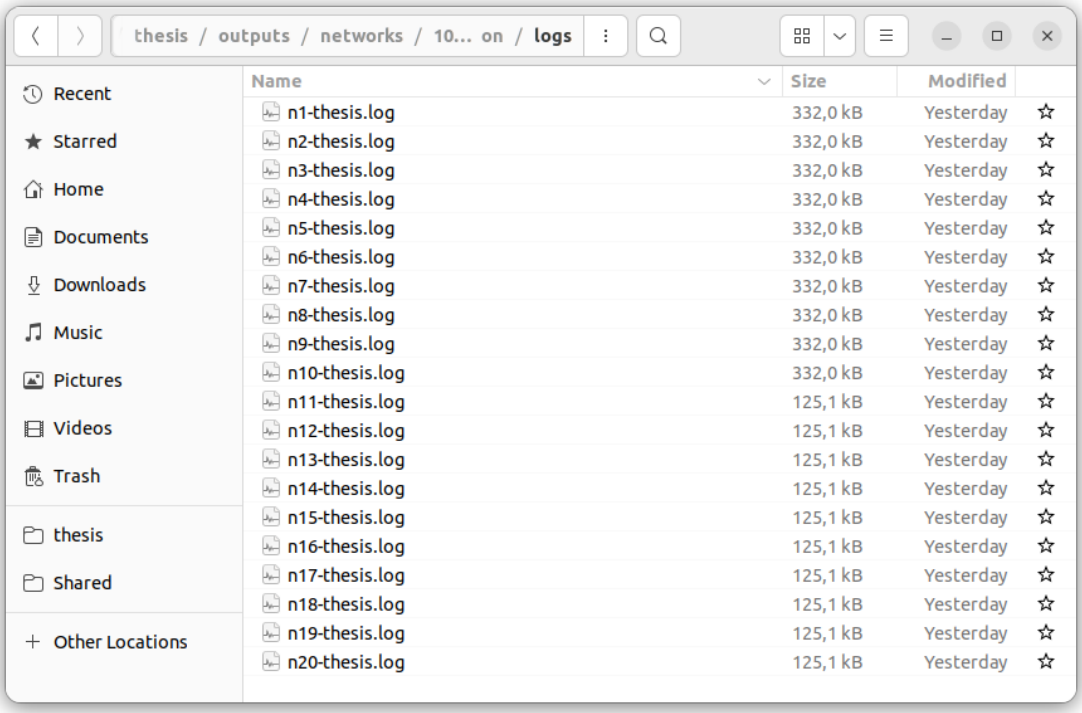


Fig. 2.22: Directorio dentro de la configuración de la red, donde se encuentran los *logs* personalizados generados por los nodos.

muy elevados. Nos interesa saber si la herramienta que utilizamos para emular las redes maneja correctamente estos cambios de latencia en las conexiones para asegurarnos de que nuestra experimentación pueda realizarse de manera correcta.

Para esto, utilizamos el sistema de los puntos anteriores, quitando la inyección de transacciones, pero aplicando demoras de 200 ms, 1,5 s y 150 s en los enlaces entre n1 y n11, y entre n12 y n13.

Dado que `SherlockFog` no nos permite cortar enlaces ya creados, podemos agregar una demora muy grande en el mismo para simular el mismo comportamiento que un enlace cortado: nos interesa que los enlaces sean abortados por el protocolo de `Algorand` (basado en *WebSocket*) y que todos los paquetes enviados a través del enlace lleguen sumamente tarde como para ser procesados. Por este motivo, consideramos el valor de 150 s como suficientemente alto para representar este comportamiento. Si bien las conexiones TCP existentes sobre ese enlace pueden persistir al agregar la demora, los paquetes enviados llegarán sumamente tarde y nunca serán procesados por el cliente. Las nuevas conexiones fallarán por este *delay* y no permitirá establecerlas.

Más allá de estos límites técnicos de las capas inferiores de comunicación, lo que nos importa es que al aplicar el corte simulado nuestro cliente `Algorand` no reciba los mensajes del otro lado del enlace afectado. Con lo cual, nuestras pruebas si bien van a verificar el nivel de transporte, también vamos a enfocarnos en analizar a nivel de aplicación que no lleguen esos mensajes.

Sistema:

- **Nodos:** 10 (n1 a n10)

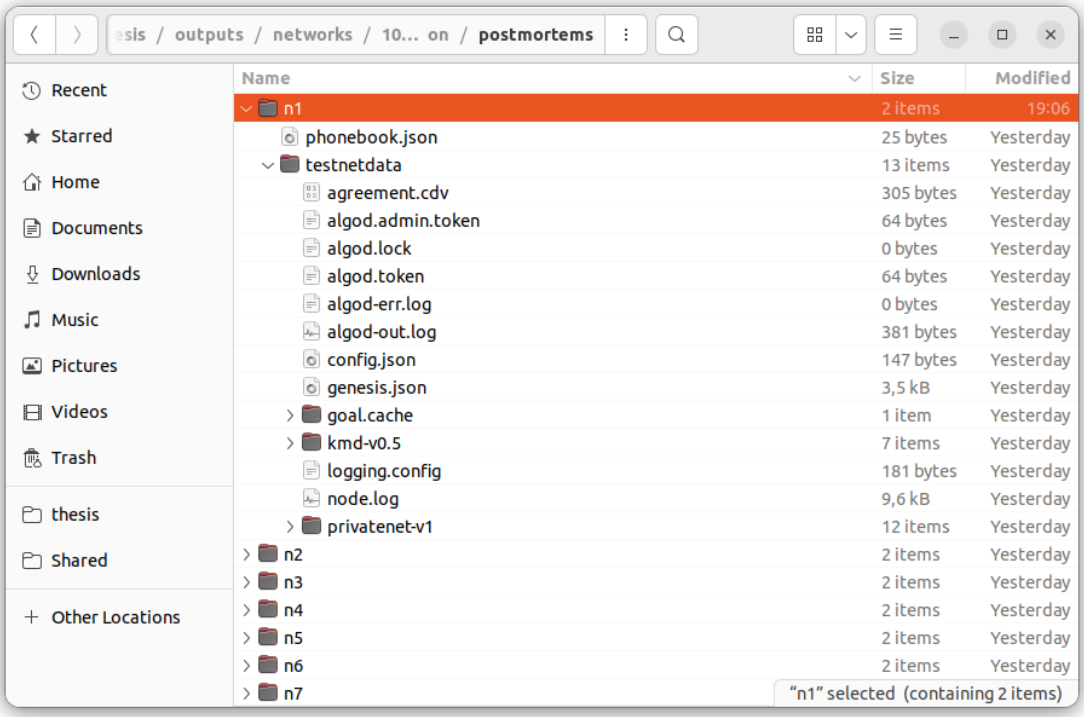


Fig. 2.23: Directorio dentro de la configuración de la red, donde se encuentran los post-mortem de los nodos.

- ***Relays*:** 10 (n11 a n20)
- ***Relays* conectados por nodo:** 1 lógico, 1 físico (20 ms)
- ***Relays* conectados por *relay*:** 4 lógicos, 1 físico (20 ms)
- ***Stake*:** homogéneo
- **Tiempo de ejecución:** 25 min
- **Servidor:** local (1 servidor)
- ***Delays* aplicados (según rango de minutos)en n1/n11 y n12/n13:**
  - 0 min a 5 min: 20 ms (valor inicial)

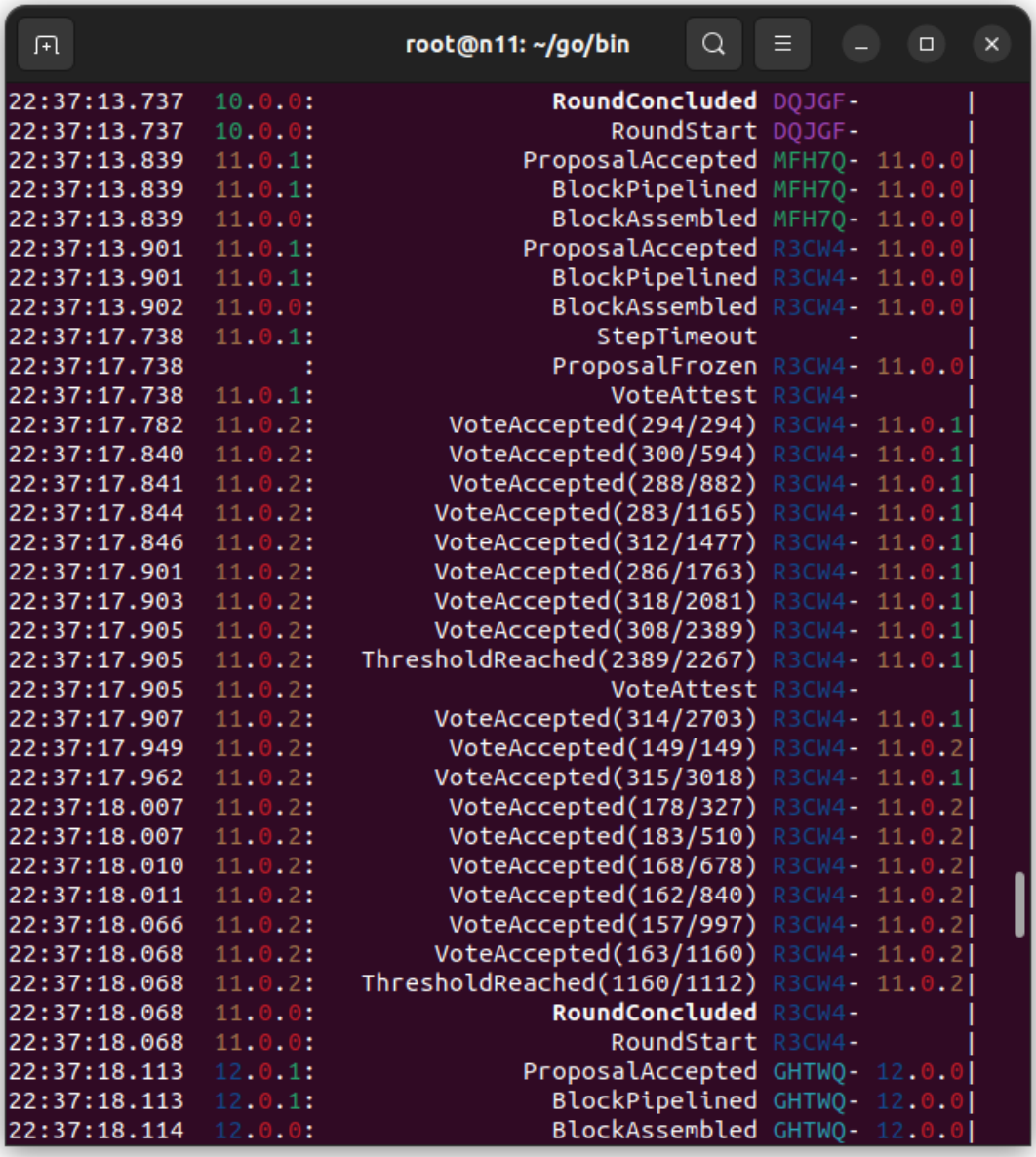


Fig. 2.24: Logs observados con `carpenter` en el comienzo del experimento en CloudLab, donde se muestra el inicio y fin de la ronda 11, durando alrededor de 4,3 s.

- 5 min a 10 min: 200 ms en enlace n1/n11 y n12/n13
- 10 min a 15 min: 1500 ms en enlace n1/n11 y n12/n13
- 15 min a 20 min: 150 s en enlace n1/n11 y n12/n13
- 20 min a 25 min: 20 ms en enlace n1/n11 y n12/n13

La idea será entonces ejecutar el experimento, ingresar en los nodos afectados y verificar que la demora en el enlace es la establecida. Es importante que, debido a la arquitectura en forma de anillo de la red, no midamos la demora hacia el nodo a través de otro enlace que no sea el que fue afectado directamente (dado que para cualquier par de *relays* existen

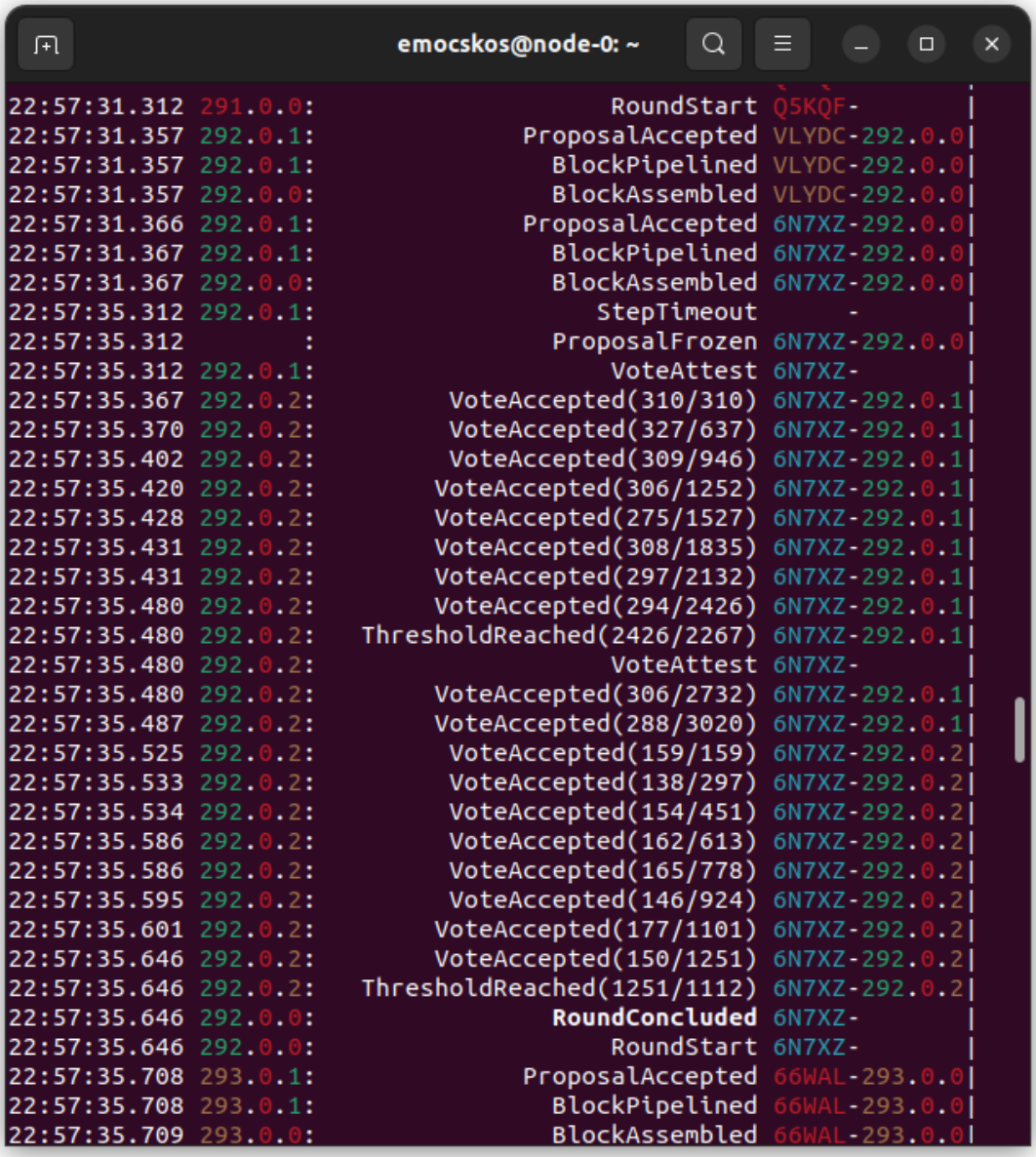


Fig. 2.25: Logs observados con `carpenter` más adelante en el experimento en CloudLab, donde se muestra el inicio y fin de la ronda 292, durando alrededor de 4,3 s.

dos caminos para conectarse entre ellos).

Entonces:

Esperamos: que en cada aplicación de *delay* a partir del *script* de `SherlockFog`, la

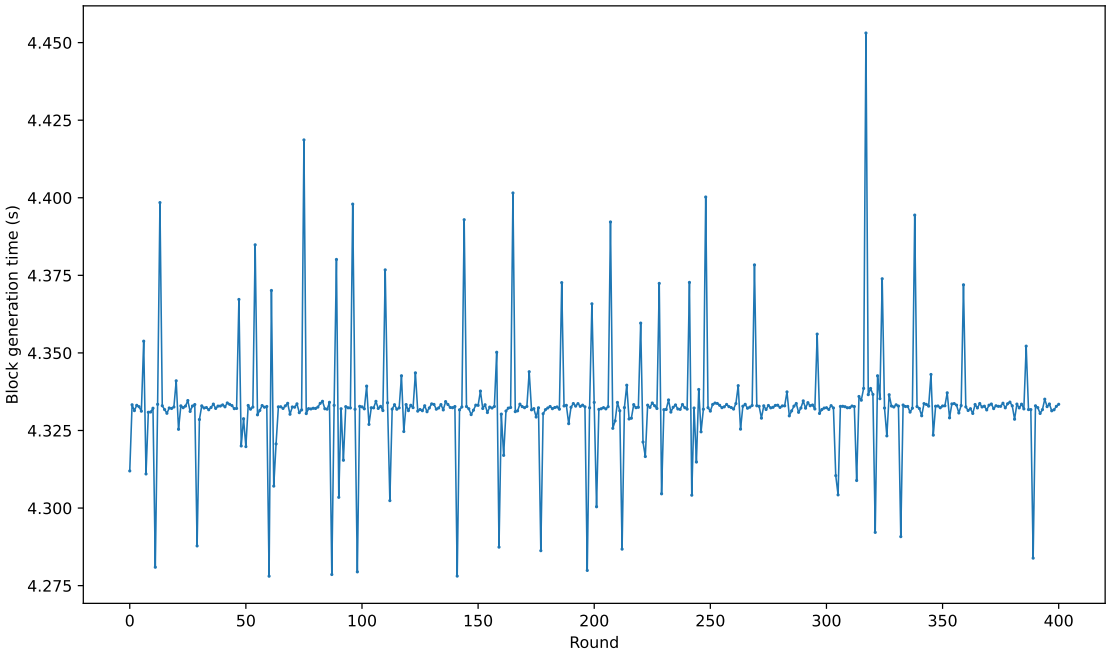


Fig. 2.26: Tiempo de generación para cada bloque del experimento en CloudLab.

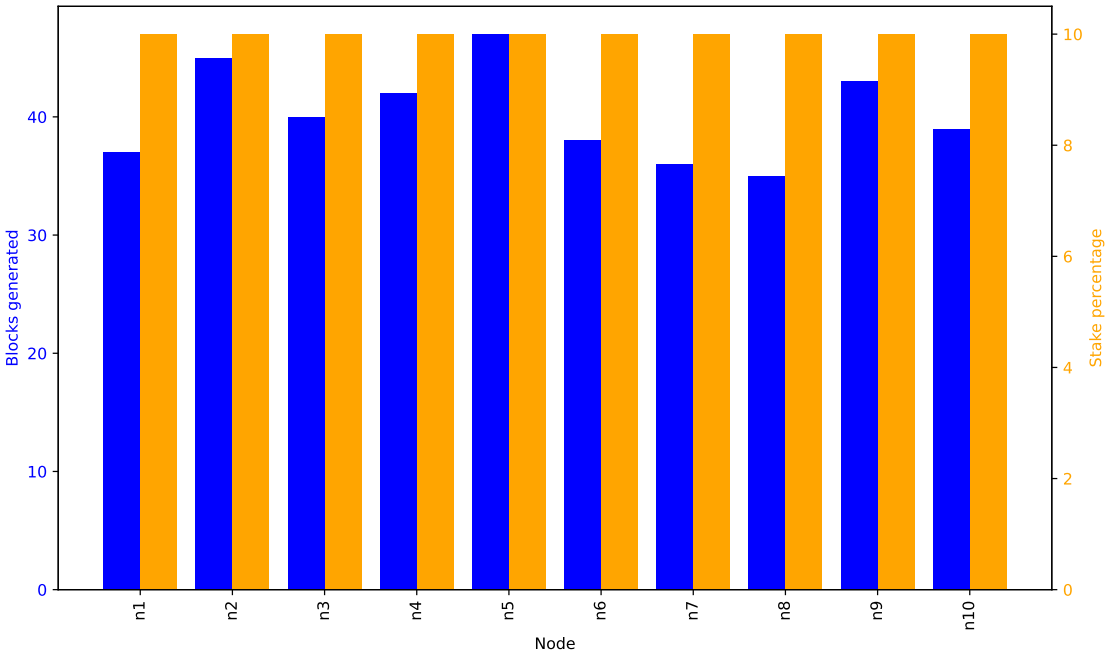


Fig. 2.27: Cantidad de bloques propuestos por nodo, junto a su *stake*, para experimento en CloudLab.

demora en los mismos se corresponda con el valor elegido en él.

Corremos: corremos el experimento, nos conectamos a los nodos n1, n11, n12 y n13 y probamos la demora entre ellos para los enlaces afectados, corriendo la herramienta `ping` para ver el valor real de la demora.

Verificamos:

Validamos las demoras dentro de cada intervalo aplicado. El valor inicial es de 20 ms en todos los enlaces.

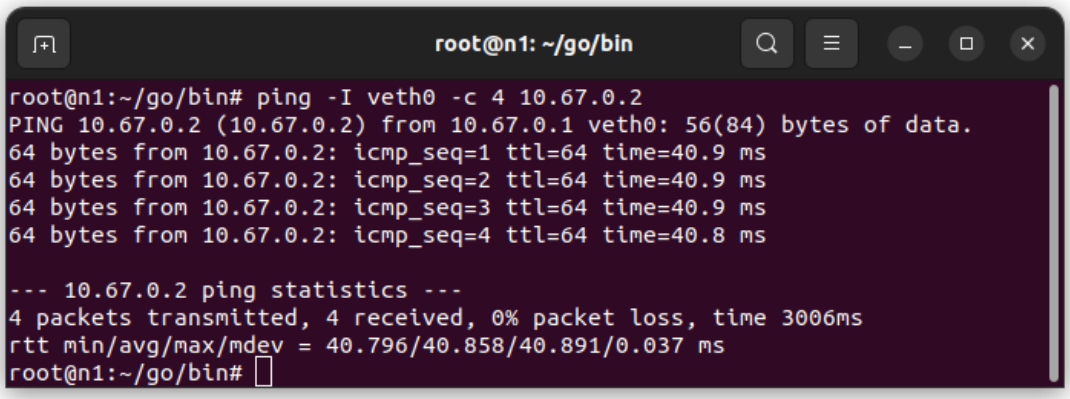


Fig. 2.28: Respuesta de comando ping desde el nodo n1 al *relay* n11 en enlace con demora de 20 ms, obteniendo un RTT de 40 ms aproximadamente.

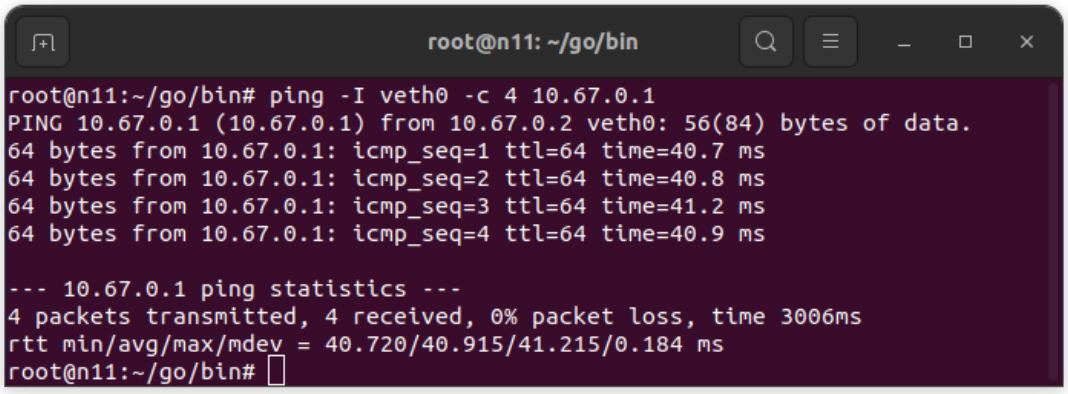


Fig. 2.29: Respuesta de comando ping desde el nodo n11 al *relay* n1 en enlace con demora de 20 ms, obteniendo un RTT de 40 ms aproximadamente.

En el enlace entre el nodo n1 y el *relay* n11 medimos un tiempo de ida y vuelta de aproximadamente 40 ms (en las imágenes 2.28 y 2.29). Esto se corresponde con el *delay* inicial del enlace: 20 ms.

Según las salidas de las figuras 2.30 y 2.31, en el enlace entre los *relays* n12 y n13 presenta el mismo comportamiento, observando nuevamente una demora dentro de los 20 ms.

Para el segundo *delay*, aplicamos una demora de 200 ms en estos enlaces. Presentamos la respuesta de uno de los nodos de cada enlace para abreviar esta sección, habiendo obtenido el mismo resultado en ambos extremos de la experiencia.

Con la demora de 200 ms en cada enlace, vemos en las imágenes 2.32 y 2.33 un RTT de aproximadamente 400 ms, lo cual valida que `SherlockFog` está aplicando el *delay* correctamente.

Al aplicar un *delay* de 1500 ms en los enlaces, obtenemos según lo visto en las figuras 2.34 y 2.35 correctamente un valor de RTT de 3000 ms. Veamos entonces el comportamiento con un *delay* mucho más grande, que simula un corte en la conexión.

Con el tiempo de espera predeterminado de la herramienta `ping`, la misma abandona la espera de una respuesta en base a la demora del enlace (visto en los gráficos 2.36 y 2.37). Probamos incrementando ese valor a 500 s, lo cual debiera aguardar el tiempo suficiente para recibir la respuesta del otro nodo. Vemos el resultado en la figura 2.38.

Con esta espera, logramos capturar la respuesta, demorando 300 s en llegar. Esto valida

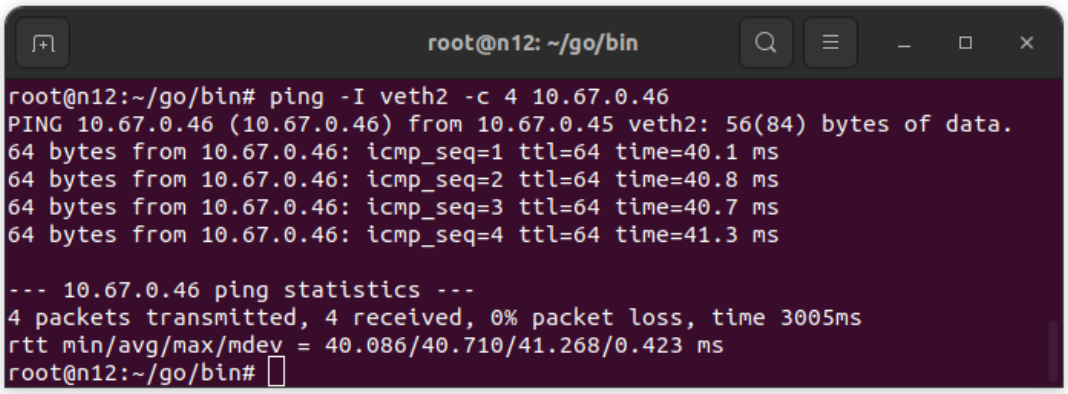


Fig. 2.30: Respuesta de comando ping desde el nodo n12 al *relay* n13 en enlace con demora de 20 ms, obteniendo un RTT de 40 ms aproximadamente.

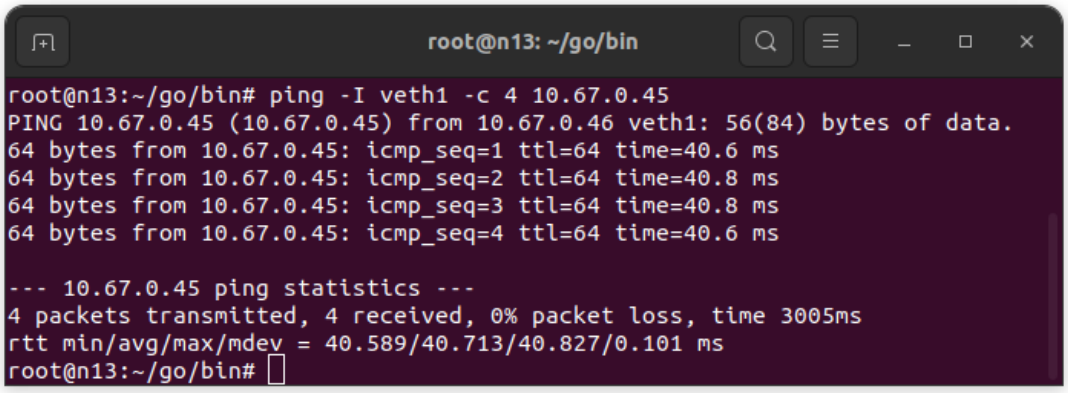


Fig. 2.31: Respuesta de comando ping desde el nodo n13 al *relay* n12 en enlace con demora de 20 ms, obteniendo un RTT de 40 ms aproximadamente.

que el *delay* aplicado en el enlace fue de 150 s.

Por otro lado, validamos que la conexión entre el cliente del nodo n1 y el *relay* n11 deja de funcionar tanto a nivel de transporte como de aplicación, sabiendo que solo este enlace físico emulado conecta al nodo n1 con el resto de la red. Chequeamos las conexiones del cliente antes del *delay*, durante y luego del mismo, observando lo siguiente:

Vemos en las imágenes 2.39 y 2.40 que al aplicar el *delay*, la conexión TCP entre los clientes recibe un fuerte impacto en el procesamiento de los paquetes, logrando también imposibilitar el establecimiento de nuevas conexiones ya que son abortadas luego del estado de *SYN_SENT* por *timeout*.

A su vez, revisamos el comportamiento a nivel de aplicación para el cliente de `Algorand`.

En la figura 2.41, el nodo n1 no logra avanzar a partir de la aplicación de la demora que simula el corte en el enlace entre él y el *relay* n11.

Una vez que volvemos a configurar los enlaces en 20 ms como al comienzo de la experiencia, podemos validar que vuelven a los valores observados en ese momento para las

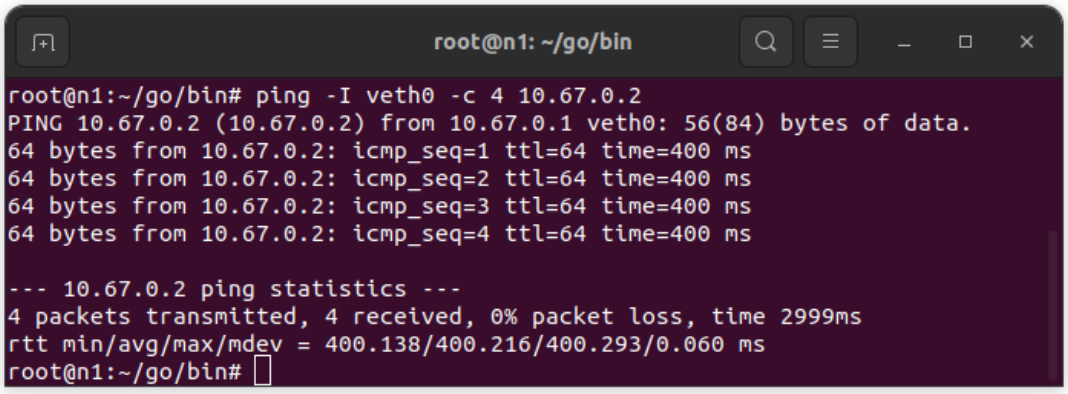


Fig. 2.32: Respuesta de comando ping desde el nodo n1 al *relay* n11 en enlace con demora de 200 ms, obteniendo un *RTT* de 400 ms aproximadamente.

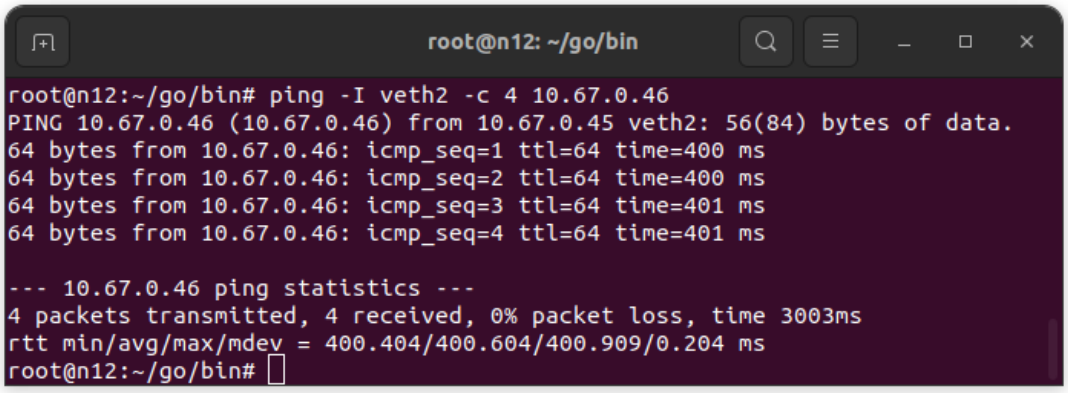


Fig. 2.33: Respuesta de comando ping desde el nodo n12 al *relay* n13 en enlace con demora de 200 ms, obteniendo un *RTT* de 400 ms aproximadamente.

figuras 2.42 y 2.43.

En efecto, la demora entre los nodos vuelve a ser la original. En cuanto al nodo n1, podemos revisar el estado del cliente al momento que se restablece el enlace en la imagen 2.44.

El nodo n1 se reincorpora al avance de la red cuando se restablece el enlace a su latencia original, viendo en la información de `carpenter` todas las rondas que pudo recuperar, y a partir de ese momento vuelve a la normalidad. Vemos también en la figura 2.45 el conjunto de mensajes recibidos por n1 en el tiempo, donde para las primeras demoras no hay un impacto en la cantidad recibida, pero que apenas se simula el corte en el enlace, deja de recibir mensajes.

Con estos experimentos pudimos validar que `SherlockFog` aplica las demoras correspondientes cuando las configuramos en el archivo *fog* de la experiencia, y que la comunicación se interrumpe con el valor de 150 s de demora en los enlaces.

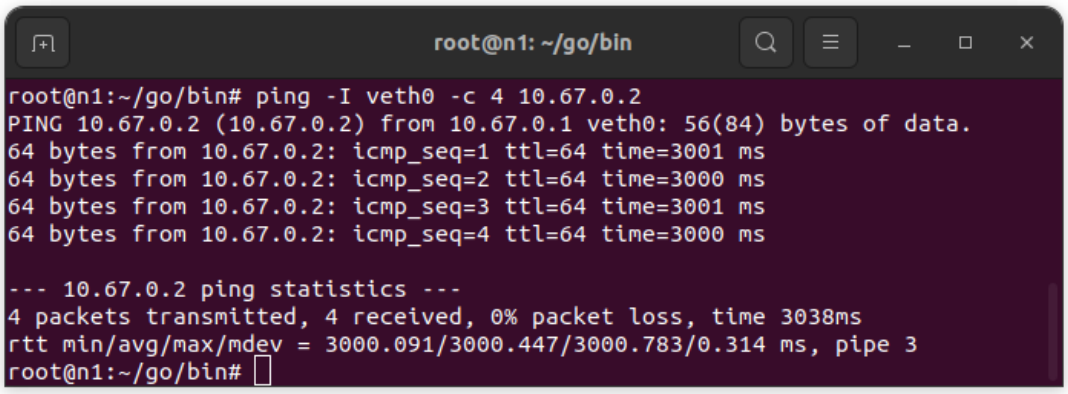


Fig. 2.34: Respuesta de comando ping desde el nodo n1 al *relay* n11 en enlace con demora de 1500 ms, obteniendo un RTT de 3000 ms aproximadamente.

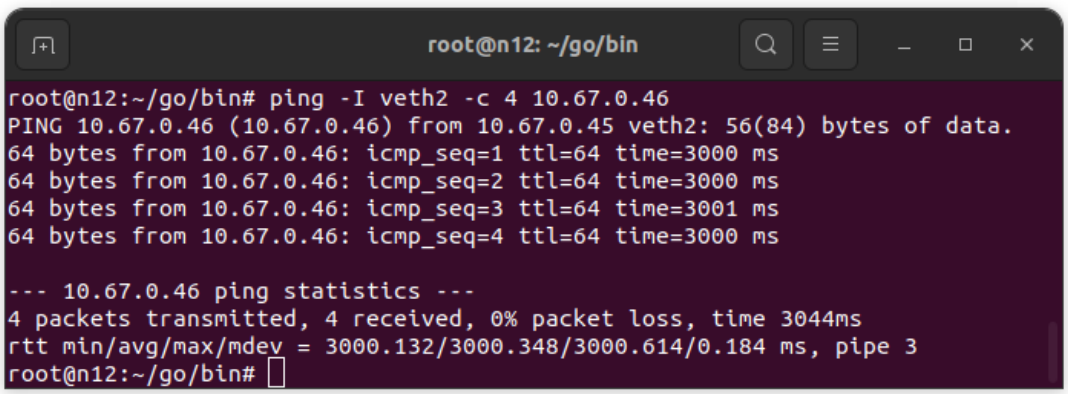


Fig. 2.35: Respuesta de comando ping desde el nodo n12 al *relay* n13 en enlace con demora de 1500 ms, obteniendo un RTT de 3000 ms aproximadamente.

2.8.13. *"Nuestro módulo en el cliente de `Algorand` captura todos los mensajes recibidos de los demás nodos"*

Para gran parte de nuestro trabajo nos va a interesar poder capturar el flujo de mensajes que recibe un *relay* durante la ejecución de la red. Con este fin, agregamos modificaciones en el cliente de manera que cada vez que se recibe un mensaje de otro nodo, este evento se guarda en el log personalizado de este trabajo y así podemos hacer un seguimiento de toda la información que recibe este nodo.

Nos interesa verificar que nuestras modificaciones son capaces de reconocer todos los tipos de mensajes que recibe el nodo (luego analizaremos cuáles son estos tipos y sus distribuciones), sin pérdida de información. Tenemos seguridad de que debe ser así ya que este código lo agregamos en donde el cliente maneja la recepción de todos los mensajes del sistema, con lo cual no debe haber otro canal de comunicación que pase por otro lado. Igualmente, para estar seguros de esto, planteamos un experimento donde vamos a capturar todos los mensajes recibidos tanto con nuestro código, como con una herramienta

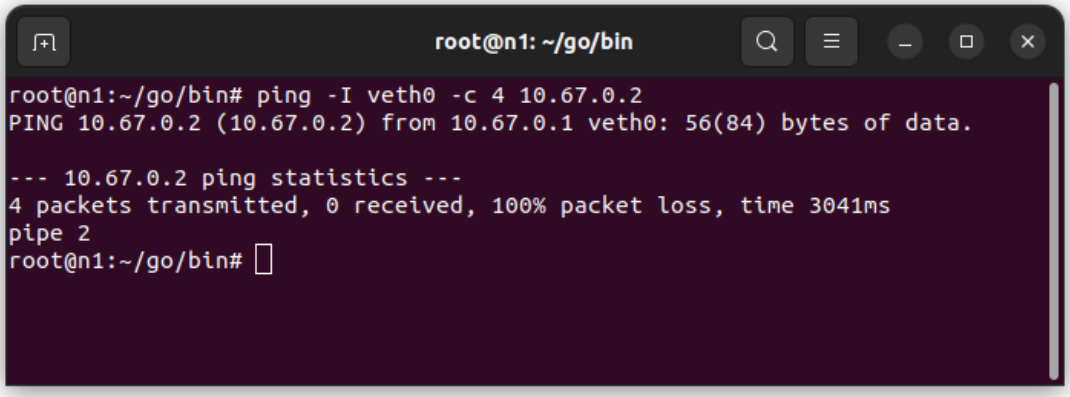


Fig. 2.36: Respuesta de comando ping desde el nodo n1 al *relay* n11 en enlace con demora de 150 s, fallando la respuesta con el tiempo predeterminado de espera.

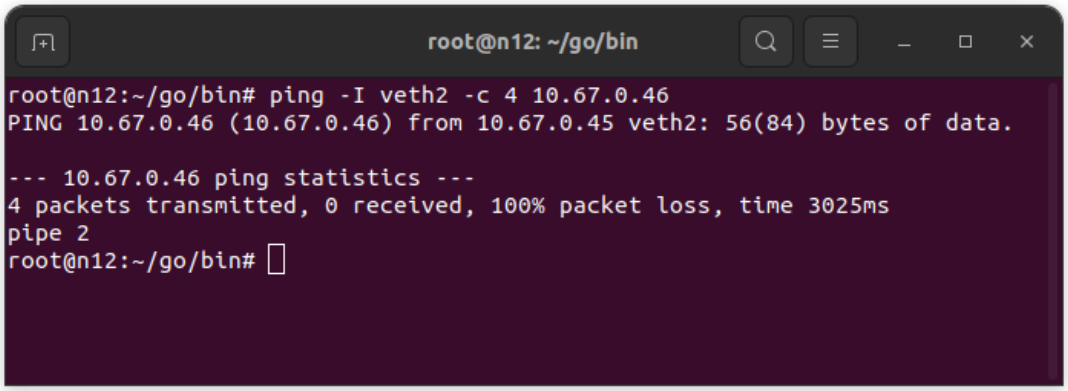


Fig. 2.37: Respuesta de comando ping desde el nodo n12 al *relay* n13 en enlace con demora de 150 s, fallando la respuesta con el tiempo predeterminado de espera.

que capture el tráfico de red y nos permita luego analizar los paquetes recibidos.

Utilizamos nuevamente el mismo sistema que en los primeros puntos de esta validación:

Sistema:

- **Nodos:** 10 (n1 a n10)
- ***Relays*:** 10 (n11 a n20)
- ***Relays* conectados por nodo:** 1 lógico, 1 físico (20 ms)

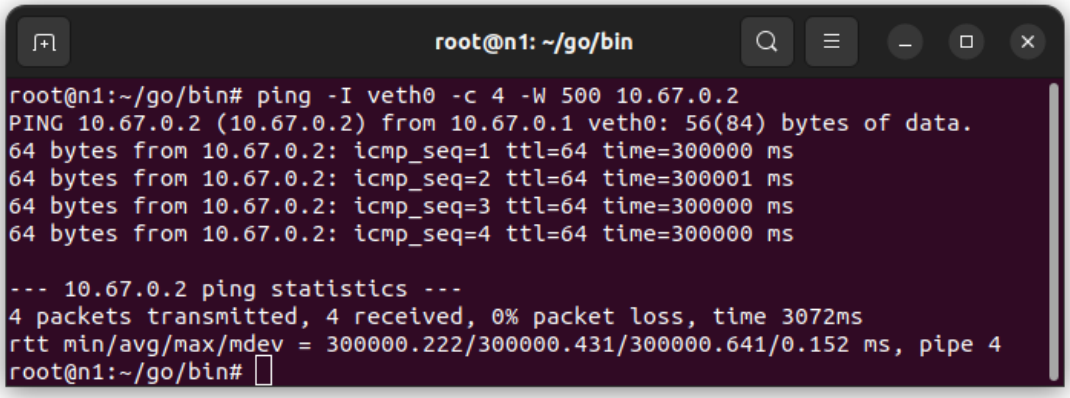


Fig. 2.38: Respuesta de comando ping desde el nodo n1 al *relay* n11 en enlace con demora de 150 s, agregando espera de hasta 500 s para la respuesta y así recibiéndola luego de 300 s de *RTT*.

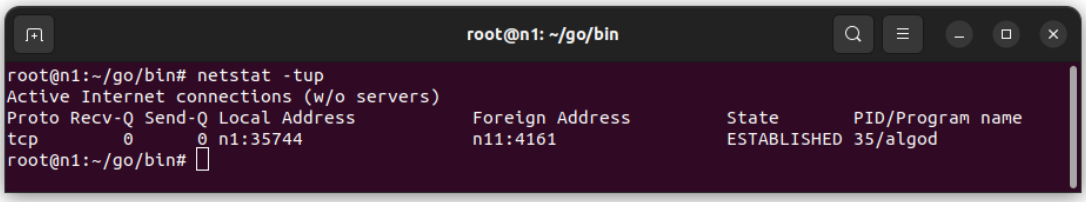


Fig. 2.39: Respuesta de comando netstat en el nodo n1 con el listado de las conexiones activas antes de aplicar el *delay* de 150 s. El cliente permanece conectado al *relay* n11 sin problemas.

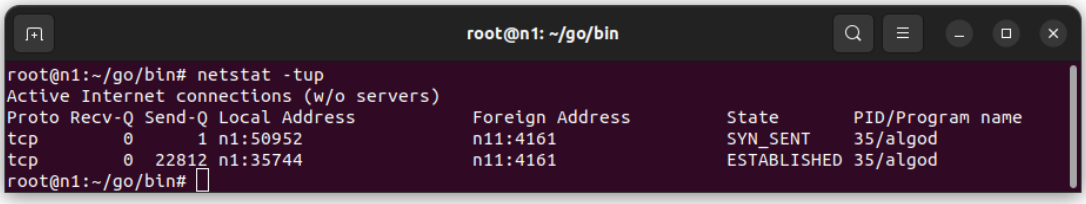


Fig. 2.40: Respuesta de comando netstat en el nodo n1 con el listado de las conexiones activas durante el *delay* de 150 s: el cliente permanece conectado al *relay*, pero la cola de envío se encuentra saturada de mensajes aguardando recibir confirmación de recepción del lado del *relay* n11, y las nuevas conexiones generadas por el cliente quedan en estado *SYN_SENT* al no poder establecer la conexión por la demora.

- ***Relays* conectados por *relay*:** 4 lógicos, 1 físico (30 ms)
- ***Stake*:** homogéneo
- **Tiempo de ejecución:** 30 min
- **Servidor:** local (1 servidor)

Para capturar los paquetes que recibe el *relay* a analizar, utilizamos `tcpdump`, ejecutándolo para que capture todos los paquetes recibidos en todas las interfaces locales.

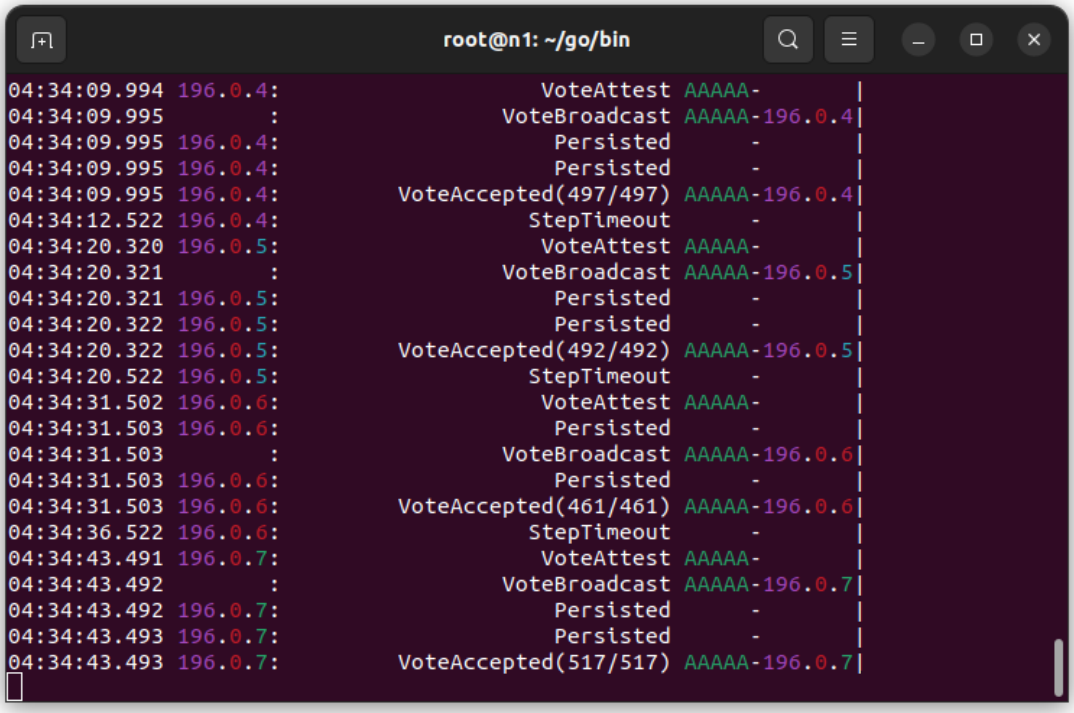


Fig. 2.41: Estado del nodo n1 durante la aplicación del *delay* de 150 s según la aplicación *carpenter*: el nodo no avanza a partir de la última ronda conocida previa al *delay*.

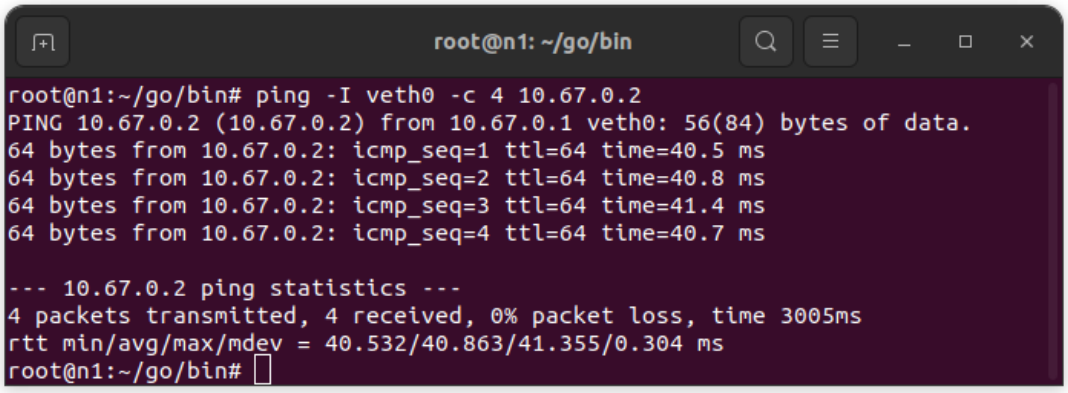


Fig. 2.42: Respuesta de comando ping desde el nodo n1 al *relay* n11 al restaurar la latencia de 20 ms en el enlace, obteniendo un RTT de 40 ms nuevamente.

En resumen, entonces:

Esperamos: que nuestro código capture la misma cantidad de mensajes para cada tipo de `Algorand` que son recibidos fehacientemente por el cliente a nivel de red.

Corremos: ejecutamos la instancia de la red, habilitando la captura de mensajes en el cliente del *relay* n11 y ejecutando `tcpdump -i any -w /nodes/capt.pcap` en él, filtrando

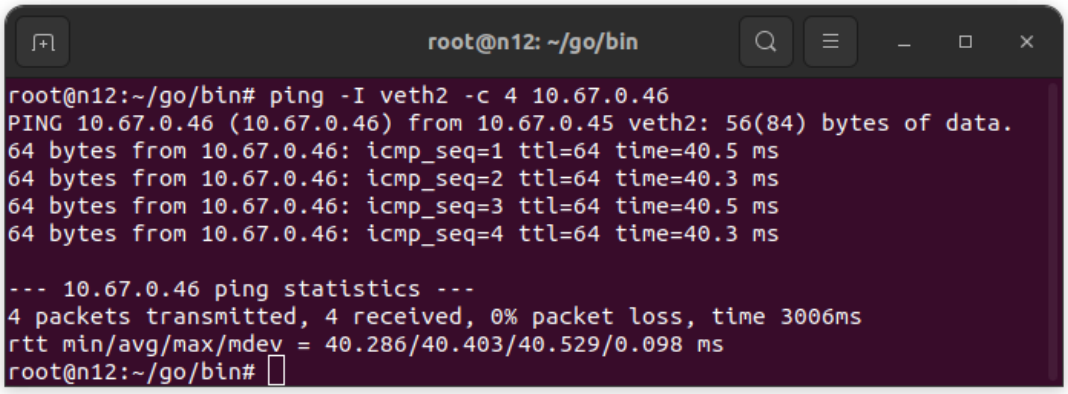


Fig. 2.43: Respuesta de comando ping desde el nodo n12 al *relay* n13 al restaurar la latencia de 20 ms en el enlace, obteniendo un RTT de 40 ms nuevamente.

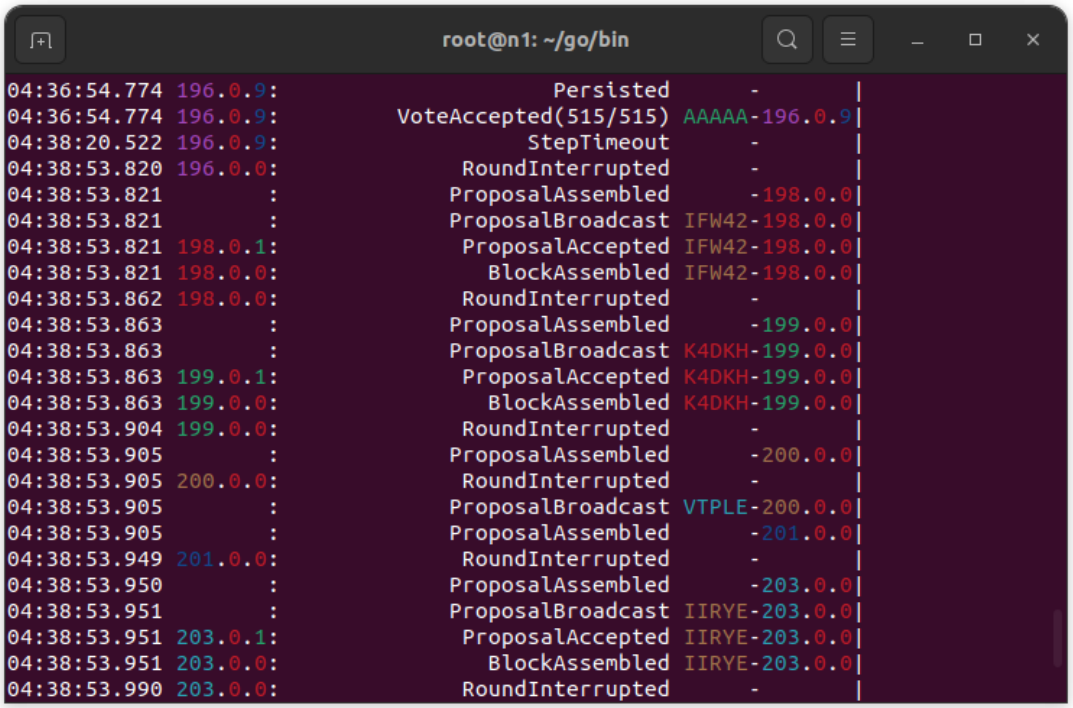


Fig. 2.44: Estado del nodo n1 luego de la aplicación del *delay* de 150 s según la aplicación *carpenter*: el nodo recupera la información de las rondas que ocurrieron mientras se encontraba detrás del *delay* y comienza a operar normalmente.

también los paquetes que tienen a nuestro nodo como destinatario o emisor.

Verificamos:

Luego de correr el experimento, utilizamos nuestras herramientas para relevar la cantidad de mensajes recibidos de cada tipo según nuestros *logs*. Aparte, analizamos la captura de paquetes realizada por `tcpdump`, pudiendo filtrar dentro del contenido de los mismos con los identificadores de cada tipo de mensaje (por ejemplo: *tx*, *PP*, *AV*).

Obtenemos los resultados expresados en la tabla 2.2.

Tipo de mensaje	**AV**	**NP**	**PP**	**UE**	**VB**	**tx**	**Total**
Código	39396	1	5474	987	9	57281	103148
Captura red	39389	1	5478	986	9	57288	103145
Diferencia	7	0	4	1	0	7	19

Tab. 2.2: Cantidad de mensajes obtenidos según tipo y total, para la captura por código propio y a través de herramientas de tráfico de red. La cantidad de mensajes registrados resulta casi idéntica, con diferencias muy pequeñas.

Vemos que para todos los casos, la cantidad de mensajes registrados por nuestro código en comparación con los capturados a nivel de red coinciden casi perfectamente, teniendo una diferencia de 19 mensajes en total, que representan un 0,000 18 % de la captura total. Entendemos que la relación entre mensajes a nivel de red versus mensajes a nivel lógico pueden discrepar por diversos motivos. Quizás alguno de estos paquetes no fue procesado por el cliente y debió ser reenviado luego por el emisor, o tal vez nuestro filtro excluyó alguna variante del contenido del mensaje que no cumplió exactamente con el mismo. De

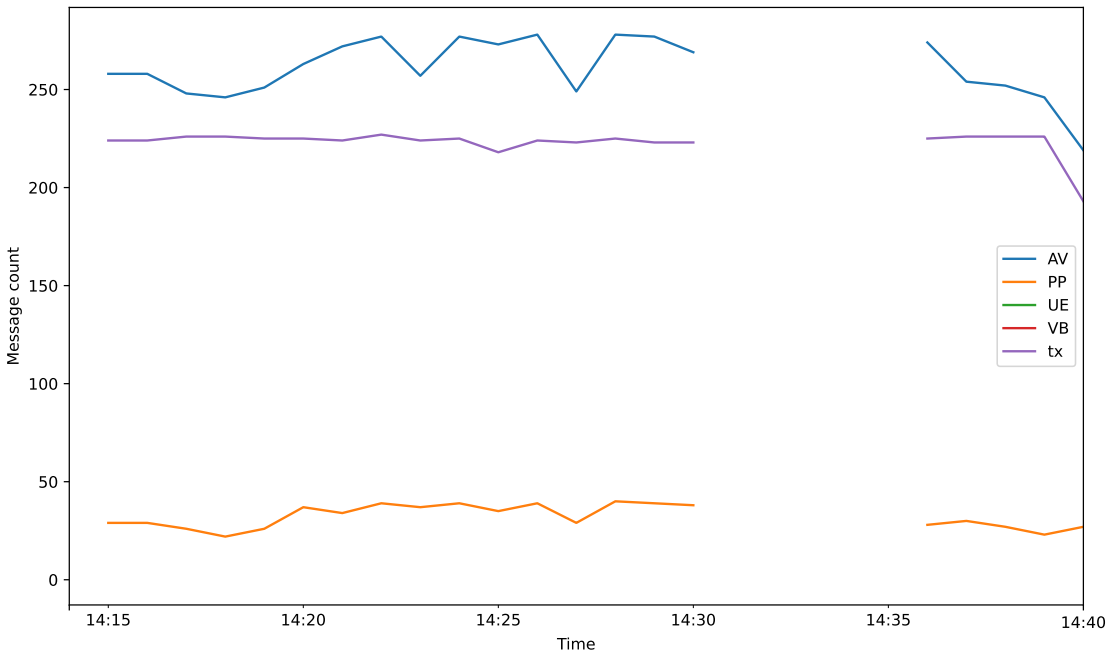


Fig. 2.45: Cantidad de mensajes recibidos por el nodo n1 durante la duración del experimento, para los tipos AV, PP, UE, VB y tx: en el momento que se aplica la demora de 150 s el nodo deja de recibir mensajes de su único *relay* conectado.

igual manera, la cantidad de diferencias que encontramos son despreciables en relación a la cantidad de mensajes analizados.

Por ello, podemos confirmar que estamos capturando correctamente los mensajes recibidos por el nodo en toda la ejecución, validando nuestro código para este fin.

2.8.14. *"Los nodos de `Algorand` se conectan a cuatro «relays» por defecto en toda su ejecución, sin cambiar por otros"*

Para poder tener el control completo del estado de la red lógica en toda la ejecución del experimento, queremos que las conexiones entre nodos y *relays* se mantengan fijas. En la realidad, tanto los nodos como los *relays* conocen el listado completo de los *relays* activos de la red y eligen aleatoriamente cuatro con los cuales intercambiar mensajes, pudiendo ocasionalmente cambiar de *relays*. Según la documentación de `Algorand` y el código del cliente, un nodo/relay cambia de *relays* a los que se conecta cuando detecta que alguno de ellos tiene problemas de conectividad o de desempeño, estando siempre alerta a otros que estén funcionando mejor (bajo criterios establecidos de ciertas medidas de *throughput*).

En la realidad, no contamos con información al respecto como para conocer con qué frecuencia efectivamente cambian de conexiones los *relays*, con lo cual decidimos realizar un experimento con escala semi-realista donde dejamos libres las conexiones de uno de los *relays* (es decir, no forzamos sus conexiones lógicas, sino que conoce a todos los *relays* del sistema y es libre de elegir aleatoriamente entre ellos), estableciendo las demás de manera fija y analizamos posteriormente todas las conexiones hechas por el primero para conocer como se distribuyen a lo largo de un experimento.

Por un lado, tomamos muestras de las conexiones realizadas por el *relay* cada cuatro segundos para analizar con qué otros *relays* se encuentra conectado a nivel de red, mientras que a su vez medimos la cantidad de *bytes* intercambiados con cada uno para también ver si esta conexión se corresponde con un volumen grande de información.

Sistema:
- **Nodos:** 150 (n1 a n150)
- ***Relays*:** 50 (n151 a n200)
- ***Relays* conectados por nodo:** 4 lógicos, 4 físicos
- ***Relays* conectados por *relay*:** 4 lógicos (excepto n151 que se deja libre), 18 físicos
- ***Stake*:** heterogéneo (siendo distribuido según la figura 2.46)
- **Tiempo de ejecución:** 10 h
- **Servidor:** CloudLab (5 servidores *d6515*)

Esperamos: que el *relay* n151 mantenga cuatro de sus pares conectados durante la mayoría del experimento, de manera que podamos asumir como correcto diagramar las redes propuestas con cuatro conexiones por nodo.

Corremos: corremos el experimento con el sistema planteado y analizamos las conexiones activas del *relay* n151 con `tcpdump` y `netstat`, para saber no sólo con qué pares se mantiene más tiempo conectado, sino también con cuáles intercambia mayor cantidad de información.

Verificamos:

Vemos que, según las figuras 2.47 y 2.48, el *relay* mantiene durante la duración del experimento cuatro *relays* principales con los que se encuentra conectado, siendo también los *relays* principales con los que intercambia información. Resulta interesante igualmente que a pesar de estar conectados a todos ellos una cantidad de tiempo similar, hay uno

de ellos con el que intercambia menos de una cuarta parte de *bytes* (el caso del n190). Esto se puede deber a que el *relay* está constantemente intentando mejorar sus conexiones, conectándose a nuevos *relays* que puedan ofrecer una mejora en cuanto a desempeño con respecto a los *relays* conectados. Como mantienen un límite de hasta cuatro *relays* con los que se interactúa en la ejecución, al incorporar uno nuevo debe eliminarse alguno de los que se estaban utilizando previamente. Este debe ser el caso del nodo n190, que se

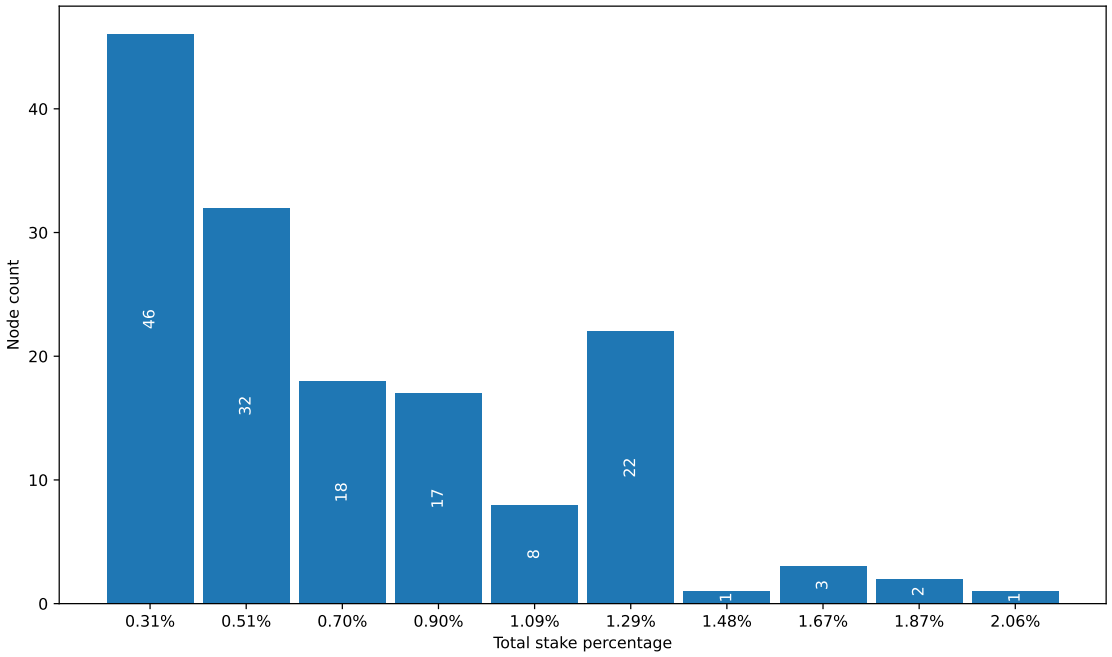


Fig. 2.46: Distribución del *stake* en el sistema según porcentaje del mismo, para 150 nodos y 50 *relays*. Se marca el valor medio de cada intervalo, con un ancho de bin del 0,15 %.

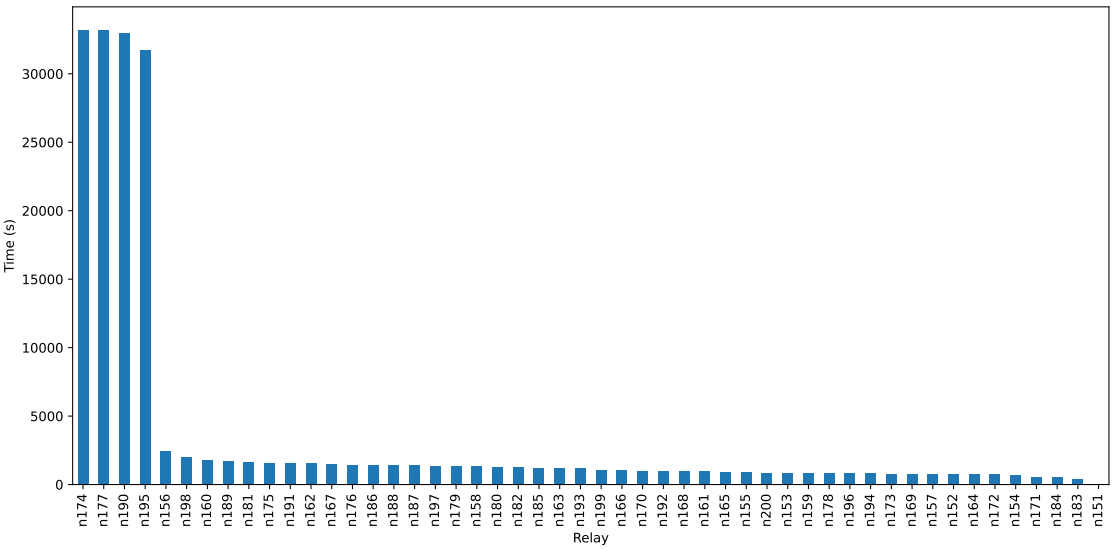


Fig. 2.47: Tiempo total de conexión del *relay* analizado a cada uno de los *relays* del sistema.

suspende mientras se prueban otros *relays*.

A pesar de este comportamiento, vemos que en toda la duración del experimento hay una diferencia sustancial en tiempo de conexión y cantidad de información intercambiada con los primeros cuatro nodos en comparación con los demás. En base a esto, para nuestros experimentos vamos a suponer arquitecturas de red donde cada *relay* tenga cuatro *relays* predefinidos para conectarse.

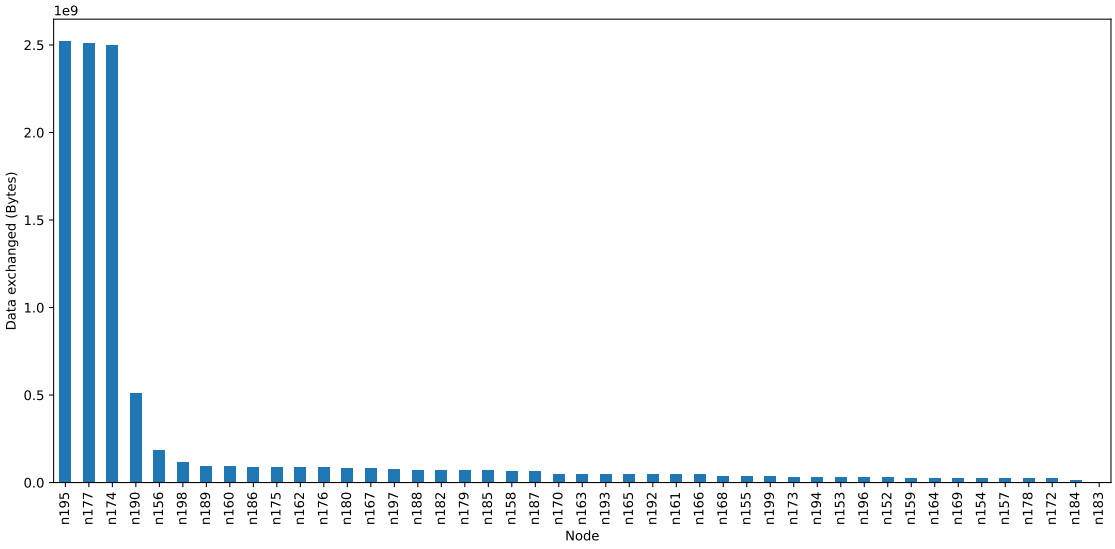


Fig. 2.48: Datos intercambiados por el *relay* analizado con cada uno de los *relays* del sistema.

3. ESTABILIDAD DE LA RED ANTE CAMBIOS EN LA TOPOLOGÍA FÍSICA

3.1. Introducción

Uno de los factores claves para cualquier red, sea descentralizada o no, es la capacidad de responder y adaptarse ante modificaciones en la topología o el funcionamiento de los integrantes de la misma. En el caso de redes centralizadas, las decisiones son tomadas por actores determinados y la reacción a estos eventos es controlada por ellos, pudiendo tomar las acciones necesarias para mantener el funcionamiento o detener la red en caso de ser necesario. Para el caso de las redes descentralizadas, no existen estos actores con mayor poder de decisión, con lo cual es una tarea compartida entre todos los nodos participantes. Esta propiedad, si bien permite tener menos susceptibilidad a ataques (en vistas de que no hay unos pocos nodos maestros), agrega complejidad a cómo enfrentar una modificación la red: todos los nodos deben tener un algoritmo de detección de fallas y recuperación, sin necesidad de esperar instrucciones de otro nodo (o al menos deberían ser capaces de determinar quién es el responsable de tomar esas decisiones).

En esta sección buscamos analizar esta capacidad de reconocimiento de interferencias en la red y adaptabilidad en la red de `Algorand`. Como fue descrito previamente, podemos correr instancias privadas de la red en un ambiente controlado donde emulamos una topología física, pudiendo agregar latencias entre los enlaces durante la ejecución del experimento. Esto nos permite hacer el seguimiento de la red durante etapas sin interferencias, verificar su correcto funcionamiento, y agregar eventos que la alteren en distinto grado.

Como interés principal, queremos ver bajo qué condiciones la red se degrada o se detiene, aplicando cortes o latencias en los enlaces para provocarlas y verificando el tiempo de recuperación necesario cuando éstas son removidas. Con lo visto previamente de la arquitectura de red de `Algorand`, notamos como crítica la estructura de *relays*, responsable de compartir todos los mensajes generados por los nodos participantes del consenso. Si estos sufrieran ataques, podría afectarse toda la conectividad de la red y así afectar el correcto funcionamiento de la misma. Estos *relays* son públicos, de manera que resulta trivial encontrar su ubicación e intentar un ataque (a diferencia de los nodos que participan del consenso).

A su vez, buscamos conocer la dificultad de generar un ataque que provoque este efecto, entendiendo que dependerá del tiempo de *delay* necesario para esto y la cantidad de nodos que debemos atacar para que suceda. Idealmente, debería ser muy costoso para un atacante detener la red o degradarla al nivel de volverla inútil (o peor aún, lograr que se detenga y no pueda resumir el progreso).

Con estos fines, planteamos distintas topologías de red para experimentar de manera incremental, comenzando con una red lineal, luego con una red en forma de anillo y finalmente con una red distribuida en forma de grafo, la cual representa una aproximación más cercana a lo que es la red real. Todas ellas cuentan con diez nodos y diez *relays* con distribución homogénea del *stake*, pudiendo mantener un buen control de la misma. A partir de estos incrementos, vemos qué propiedades se trasladan en cada uno de los casos y qué particularidades aparecen en cada una.

3.2. Resumen de los experimentos y resultados

Planteamos ocho escenarios distintos para analizar el impacto de interrupciones y demoras en diferentes enlaces de cada red. Creamos redes de diez nodos y diez *relays*, conectando física y lógicamente cada nodo a un *relay*, e interconectando a los *relays* lógicamente a otros cuatro, pero físicamente de tres maneras distintas: trabajamos primero con tres escenarios de red lineal, luego con una de anillo y finalmente con cuatro en forma de grafo.

Las conexiones lógicas entre nodos y *relays* se mantiene igual para todos ellos. Por otro lado, el *stake* se distribuye equitativamente entre los nodos del sistema.

- Red lineal de *relays*
  - Interrupción en el medio
    - Interrumpimos la conexión de la red de *relays* por la mitad.
    - **Resultado:** la red detiene su avance y no se producen bloques durante el corte. Al levantarlo, la red tarda 82 s para recuperarse.
  - Interrupción desde un extremo
    - Interrumpimos la conexión de la red de *relays* desde un extremo, abarcando desde uno hasta tres enlaces de la red.
    - **Resultado:** con uno y dos *relays* afectados por el corte, la red no se ve afectada. Al agregar un tercero se detiene la red y no se producen bloques. Removido el corte, la red toma 52 s en recuperarse.
  - *Delays* incrementales
    - Afectamos el tercer enlace desde un extremo de la red de *relays*, quedando tres de ellos por detrás del mismo, y aumentamos la latencia del mismo de manera incremental.
    - **Resultado:** con 1 s de latencia, la red aumenta el tiempo de producción de bloques en más del 50 %. A partir de 2 s se degrada fuertemente el avance, tomando hasta 26 veces más en producir bloques. Finalmente, con más de 6 s la red detiene su avance. Reinstalada la latencia original, la red se recupera en 28 s.
- Red anillo de *relays*
  - *Delays* incrementales
    - Afectamos dos enlaces de la red de manera de aislar tres *relays* del anillo.
    - **Resultado:** observamos un comportamiento similar al escenario lineal, con 1 s de latencia, el tiempo por bloque aumenta un 50 %. Con latencias mayores se logra degradar la red, pudiendo detenerla con latencia de 7 s. Levantada la afección, se producen bloques luego de 14 s.
- Red de *relays* en clique
  - *Delays* incrementales en *relays*
    - Aplicamos *delays* incrementales en todos los enlaces físicos hacia otros *relays* de tres *relays*.

- **Resultado:** con demoras de 1 s a 6 s se ve afectado el tiempo por bloque significativamente, llegando a detenerse al elevarlo a 7 s. Al disminuirlo al valor inicial, la red se recupera en 6 s.

- *Delays* incrementales incluyendo nodos
  - Aplicamos *delays* incrementales en todos los enlaces físicos hacia otros *relays* y los nodos conectados de tres *relays*.
  - **Resultado:** con demoras de 1 s a 3 s se logra degradar la red, pudiendo detenerla con solo 4 s al afectar no solo los enlaces entre *relays*, sino también la conexión a su nodo conectado. Reinstalado el funcionamiento original, se produce un nuevo bloque luego de 12 s.

- *Delays* incrementales solo en nodos
  - Aplicamos *delays* incrementales en todos los enlaces físicos hacia los nodos conectados de tres *relays*.
  - **Resultado:** con demoras de 1 s a 6 s se aumenta el tiempo por bloque, deteniéndose a partir de los 7 s. Al levantar la demora, la red recupera su avance en 88 s.

- Nodos libres: afectando *relays* uno por uno
  - Conectamos físicamente a los nodos a todos los *relays* y los configuramos con el listado completo de *relays* (y no forzando uno solo como en los anteriores). Afectamos todos los enlaces físicos de cada *relay*, *relay* por *relay* hasta afectarlos a todos.
  - **Resultado:** a medida que se van afectando *relays*, los nodos detectan las fallas y se conectan a otros de los disponibles, sin ver cambios en el tiempo por bloque. Finalmente, al afectar a todos los *relays*, la red se detiene y una vez que se remueven las demoras, retoma su progreso en 27 s.

3.3. Red con topología física lineal: interrupción en el medio

El primer caso a analizar consiste en una red con topología lineal, donde generamos una interrupción en la conexión en la mitad de la misma. Dado que nuestra herramienta no cuenta con la posibilidad de eliminar enlaces sobre la marcha, generamos un *delay* muy grande de manera que todos los paquetes sean descartados una vez que atraviesan el enlace (lo seteamos en 150 s).

Conectamos diez *relays* en una línea, mientras que los diez nodos están conectados cada uno a uno de los *relays* (de manera que hay una relación uno a uno con ellos). Los *relays* se encuentran conectados a cuatro otros *relays* predeterminados.

Tanto en este experimento, como en los siguientes, todos los nodos y *relays* del sistema buscan el mejor camino físico hacia los demás al ser inicializados. Dado que conectamos los nodos directamente a sus *relays*, y a los *relays* con todos sus pares, siempre elegirán este camino sin saltos para crear los caminos lógicos que luego utilizará el cliente de `Algorand` para sus conexiones. De esta manera, nos aseguramos de que conocemos de antemano cuál es el camino que está eligiendo cada cliente para comunicarse con los demás actores de la red.

Sistema:

- **Nodos:** 10 (n1 a n10)

- ***Relays*:** 10 (n11 a n20)
- ***Relays* conectados por nodo:** 1 lógico, 1 físico (20 ms)
- ***Relays* conectados por *relay*:** 4 lógicos, 1 físico (30 ms)
- ***Stake*:** homogéneo
- **Tiempo de ejecución:** 30 min
- **Servidor:** local (1 servidor)
- ***Delays* aplicados (según rango de minutos):**
  - 0 min a 10 min: 30 ms (valor inicial)
  - 10 min a 20 min: 150 s en enlace n15/n16
  - 20 min a 30 min: 30 ms en enlace n15/n16

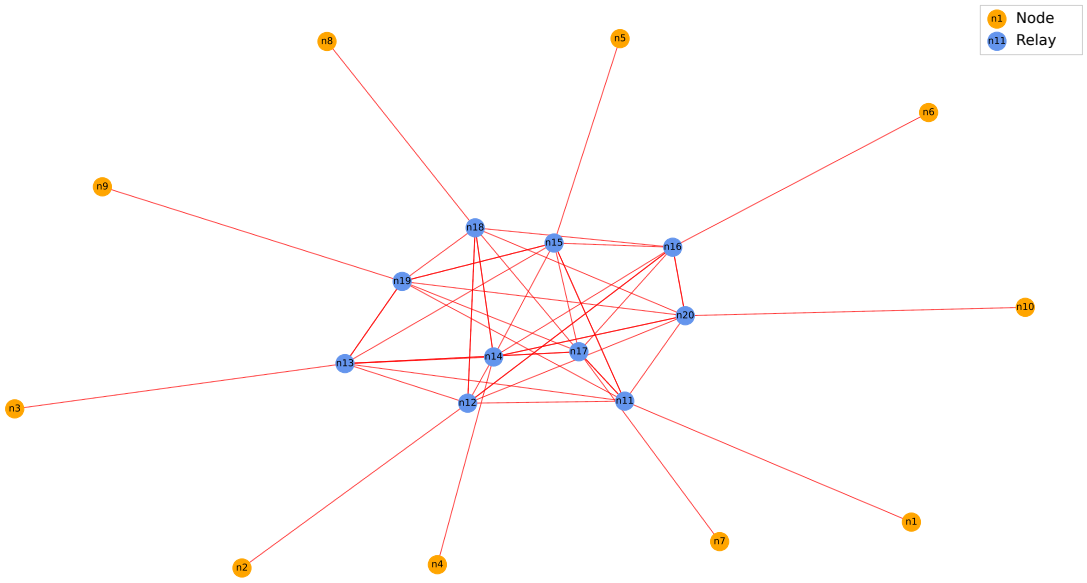


Fig. 3.1: Topología de la red lógica.

Si bien la topología lógica que utilizamos, en la imagen 3.1, es lo que esperaríamos de una red realista, la planteamos sobre una topología física con forma de línea como se observa en la imagen 3.2.

Corremos una instancia donde inicialmente la red se encuentra con las latencias sin alterar, se la deja correr durante 10 min, y luego aplicamos una latencia de 150 s en el enlace entre n15 y n16 durante 10 min para observar las consecuencias. Finalmente restablecemos la latencia inicial para analizar el tiempo de recuperación de la red y aguardamos 10 min más.

Al dividir la red a la mitad como se muestra en la imagen 3.3, dejamos la mitad del *stake* de cada lado incomunicado, con lo cual esperamos que la misma no logre avanzar.

Vemos en la figura 3.4 que desde el comienzo del experimento hasta que se aplica el *delay* se producen bloques con el mismo ritmo. Apenas comienza el *delay*, la red no puede producir más bloques dado que no consigue el quorum suficiente para avanzar (se necesita más del 80 % del *stake* activo en el consenso para poder producir un bloque). Durante los 10 min del *delay* la red no avanza. Luego, se restablece el enlace y la red logra recuperarse luego de 82 s, retomando el mismo ritmo de producción de bloques con el que contaba previamente.

Analizando la figura 3.5, el tiempo de generación de bloques se mantiene constante hasta la llegada del *delay*, donde se produce un pico ante la imposibilidad de generar el siguiente. Una vez restablecido el sistema, el tiempo vuelve a ser el anterior. Podemos apreciar en el gráfico 3.6 un detalle del tiempo por bloque aislando el pico detectado.

Verificamos en la figura 3.7 también que la producción de bloques durante el experimento resulta equitativa para la duración del mismo, siendo que todos los nodos mantienen la misma cantidad de *stake*. Veamos también que observamos el mismo comportamiento

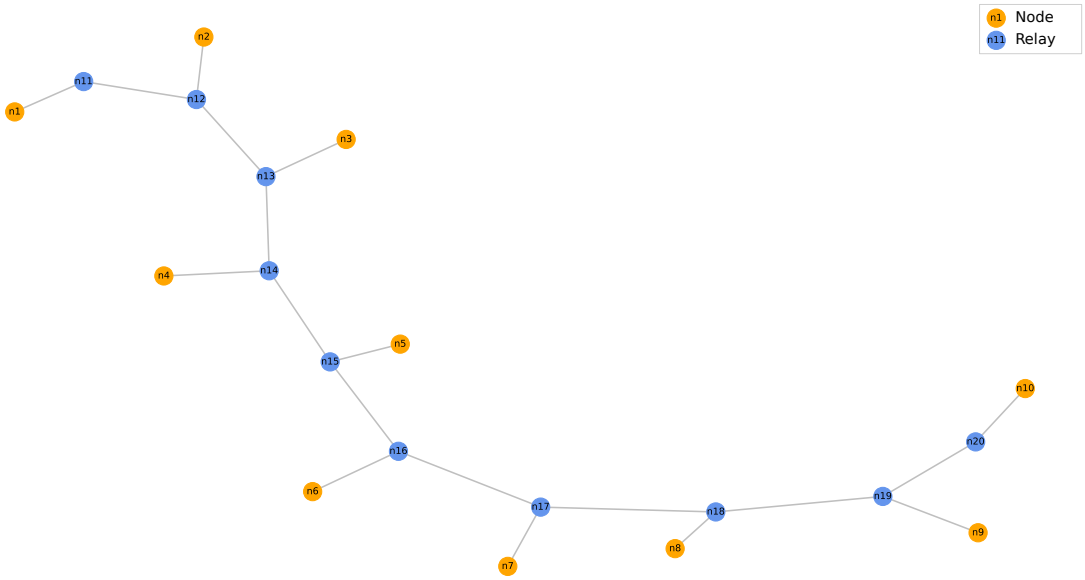


Fig. 3.2: Topología de la red física.

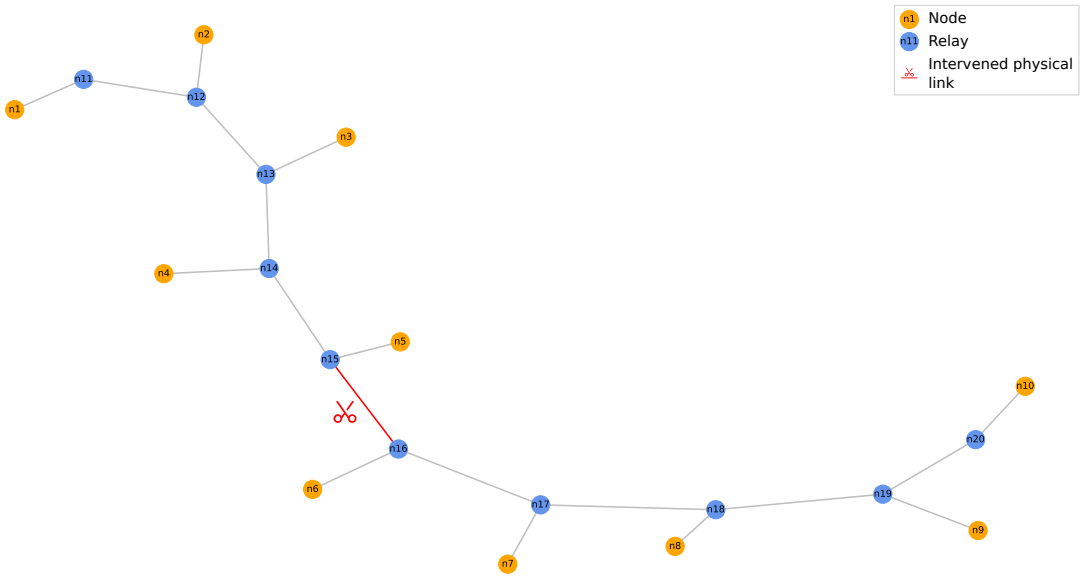


Fig. 3.3: Topología física, se muestran los enlaces afectados en la red, simulando un corte entre los nodos n15 y n16.

si nos paramos del otro lado de la red, en el nodo n17.

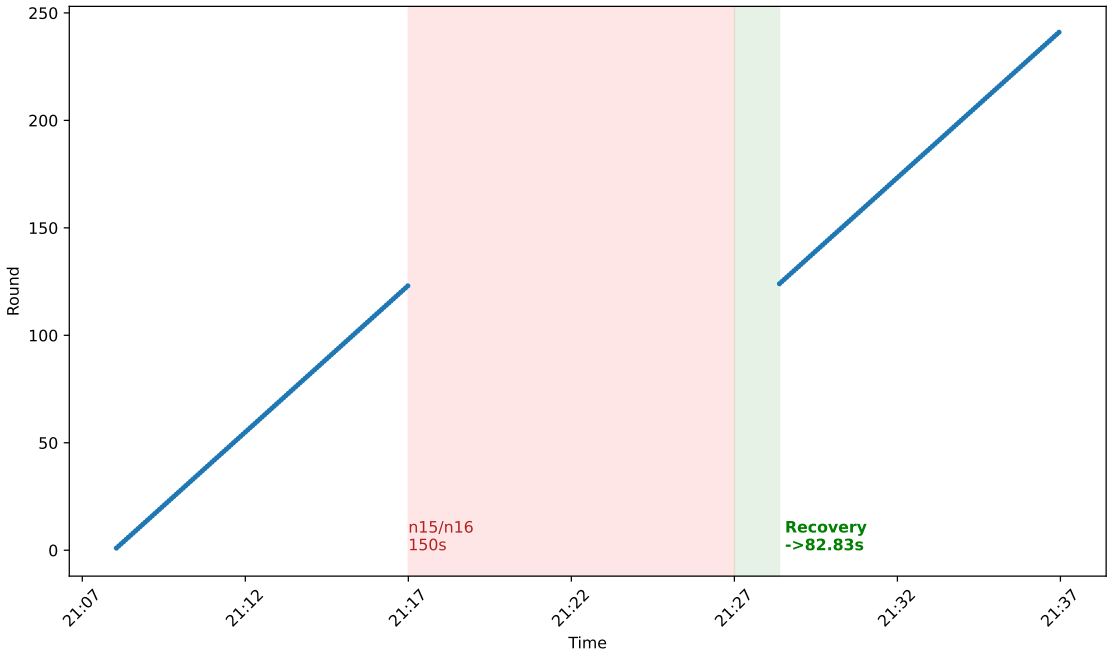


Fig. 3.4: Bloques producidos por la red en todo el experimento tomados del nodo n15, donde el área roja delimita el tiempo donde se aplica el *delay* de 150 s entre los nodos n15 y n16, mientras que el área verde se corresponde con el tiempo de recuperación de la red para retomar la producción de bloques.

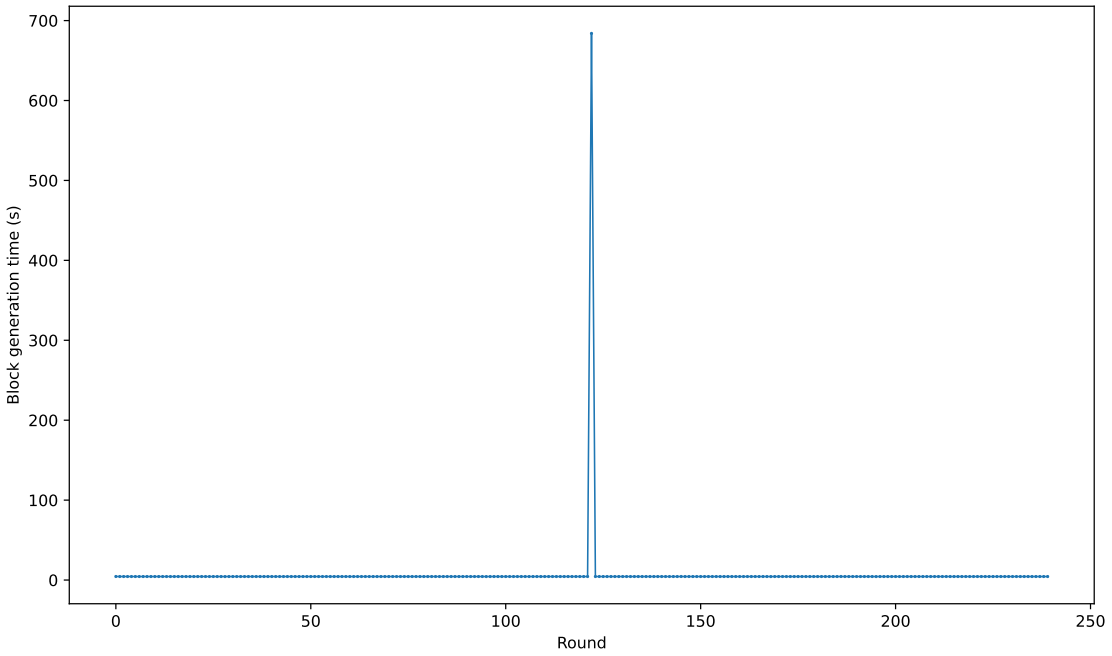


Fig. 3.5: Tiempo para la generación del bloque de cada ronda del experimento.

Siguiendo lo visto en la figura 3.8, el análisis resulta ser el mismo, con lo cual en ambos extremos de la red se detiene la producción de bloques y no avanza hasta que se levanta el *delay*.

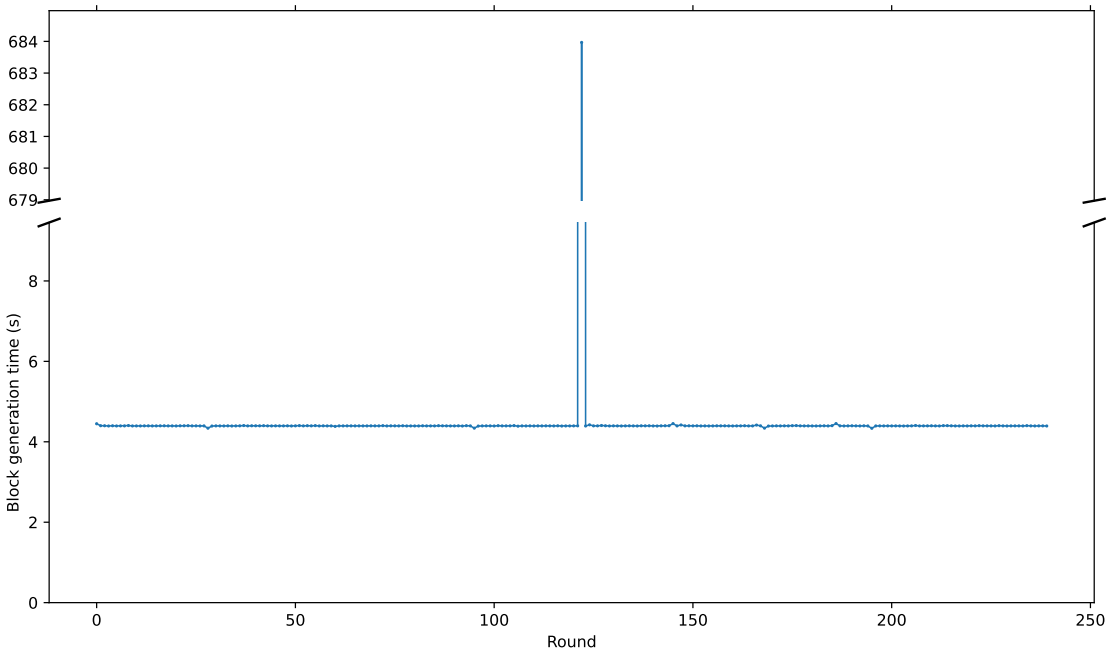


Fig. 3.6: (Detalle) Tiempo para la generación del bloque de cada ronda del experimento.

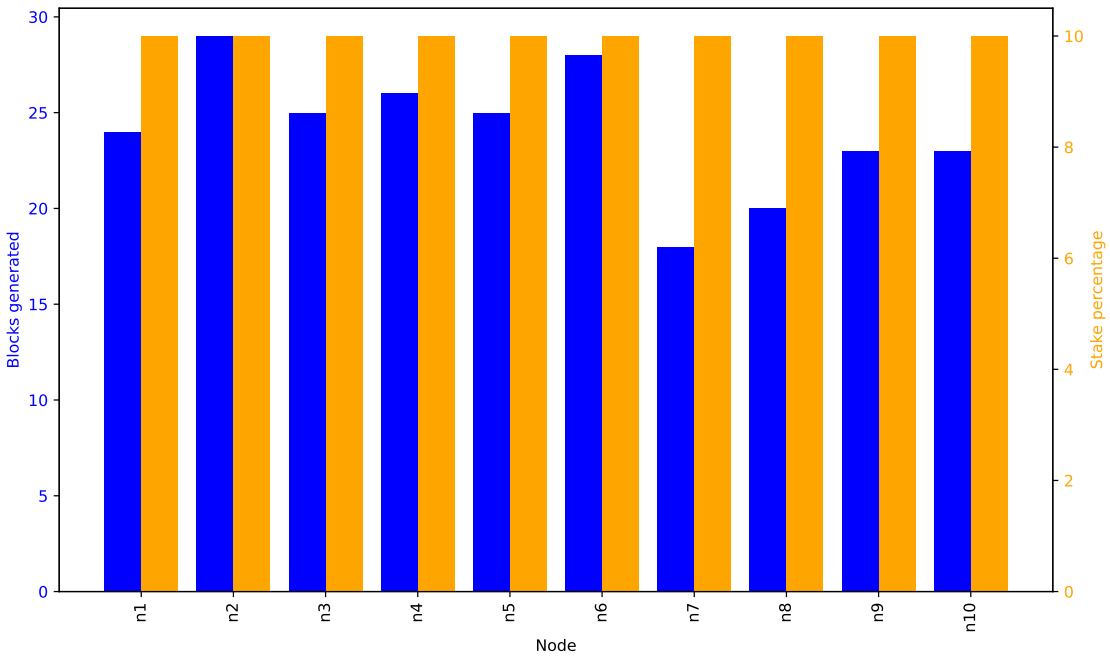


Fig. 3.7: Cantidad de bloques generados por nodo y su *stake* correspondiente.

3.4. Red con topología física lineal: interrupción desde un extremo

Nos interesa ahora conocer qué sucede si en lugar de cortar la red física por la mitad, vamos cortando un extremo de la misma incrementalmente: en cierto punto la red debería detenerse por el mismo motivo que el caso anterior.

Para esto, comenzamos nuevamente con la red en un estado inicial con avance por 10 min, y agregamos un *delay* de 150 s entre los nodos n19 y n20 (ambos *relays* del extremo de la red) por 10 min. Luego, restablecemos esa conexión por 10 min y posteriormente aplicamos el mismo *delay* entre los nodos n18 y n19. Así, vamos trasladando el *delay* entre nodos para encontrar en qué punto la red deja de operar, hasta la conexión entre los nodos n17 y n18. En la figura 3.9 se muestran estos cortes en el orden dado.

La red utilizada es la misma que en el experimento anterior, solo se modifica el tiempo de ejecución que en este caso pasa a ser de 80 min.

Sistema:

- **Nodos:** 10 (n1 a n10)
- ***Relays*:** 10 (n11 a n20)
- ***Relays* conectados por nodo:** 1 lógico, 1 físico (20 ms)
- ***Relays* conectados por *relay*:** 4 lógicos, 1 físico (30 ms)
- ***Stake*:** homogéneo
- **Tiempo de ejecución:** 80 min
- **Servidor:** local (1 servidor)
- ***Delays* aplicados (según rango de minutos):**
  - 0 min a 10 min: 30 ms (valor inicial)

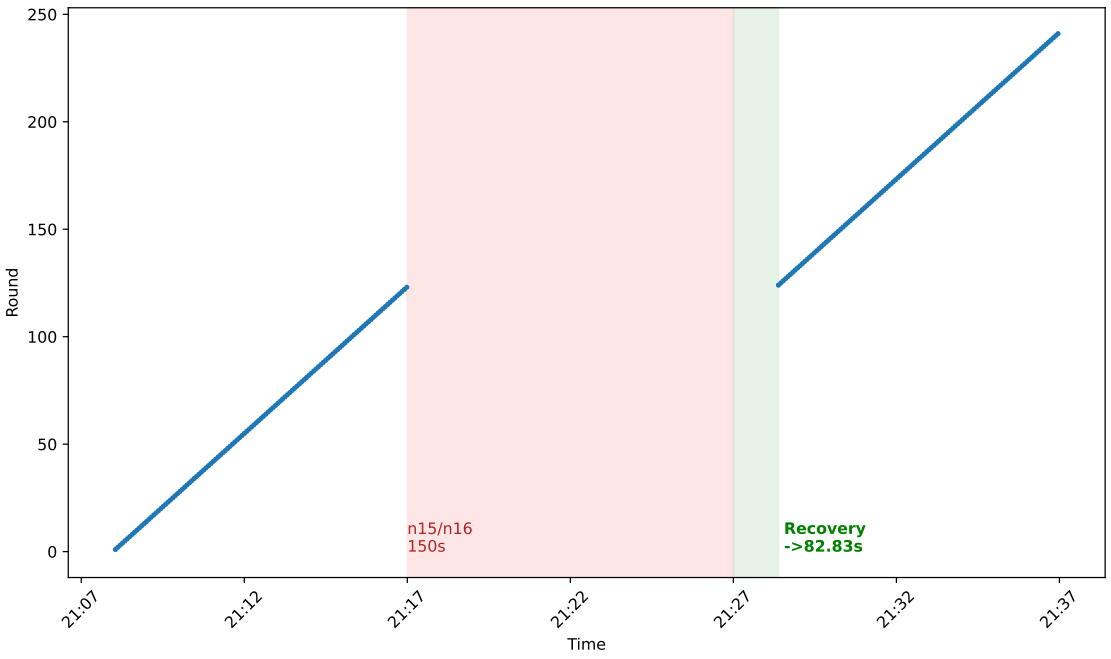


Fig. 3.8: Bloques producidos por la red en todo el experimento tomados del nodo n17, donde el área roja delimita el tiempo donde se aplica el *delay* de 150 s entre los nodos n15 y n16, mientras que el área verde se corresponde con el tiempo de recuperación de la red para retomar la producción de bloques.

- 10 min a 20 min: 150 s en enlace n19/n20 (corte I)
- 20 min a 30 min: 30 ms en enlace n19/n20
- 30 min a 40 min: 150 s en enlace n18/n19 (corte II)
- 40 min a 50 min: 30 ms en enlace n19/n20
- 50 min a 60 min: 150 s en enlace n17/n18 (corte III)
- 60 min a 80 min: 30 ms en enlace n17/n18

Tomamos los resultados graficados en la figura 3.10 y los analizamos. El primer *delay* que se aplica, entre los nodos n19 y n20, solo deja un 10 % del *stake* fuera del sistema, por lo cual se siguen produciendo bloques. Cuando se aplica el segundo *delay*, entre los nodos n18 y n19, dejamos un 20 % del *stake* del sistema aparte. Notamos una perturbación en los tiempos de generación de bloques, que se mantiene durante el tiempo posterior donde se levanta el *delay*. Sin embargo, no se frena el sistema y se producen igualmente bloques.

A diferencia de esto, cuando se aplica el *delay* entre los nodos n17 y n18, dejando por fuera un 30 % del *stake* activo, el sistema ya no puede alcanzar el quorum necesario y se dejan de producir bloques. Al restaurar el *delay* inicial, el sistema retoma la generación en cuestión de 52 s, teniendo algún sobresalto sobre la misma, pero manteniendo el ritmo que tenía al comienzo de la experiencia.

En cuanto a los tiempos de generación de bloques, vemos en la figura 3.11 un claro pico que representa el primer bloque posterior a que se levante el último *delay*. También encontramos algunos picos pequeños para los períodos donde el segundo *delay* está vigente, observables en el detalle de la figura 3.12.

Podemos pararnos sobre el nodo n19 para conocer cuáles fueron los bloques vistos por este *relay* durante la duración del experimento, siendo que debido a los *delays* quedó apartado de la porción de la red que siguió avanzando.

Para este caso, vemos en la figura 3.13 que el nodo n19 deja de observar bloques

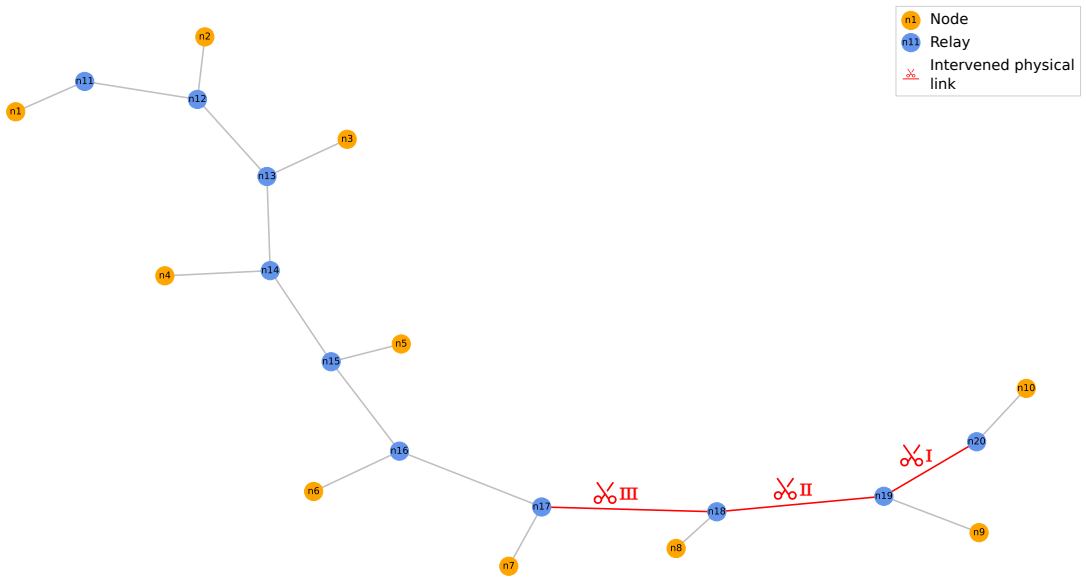


Fig. 3.9: Topología física, se muestran los enlaces afectados en la red, simulando un corte inicialmente entre los nodos n19 y n20 (corte I), luego entre n18 y n19 (corte II) y finalmente entre n17 y n18 (corte III).

durante el *delay* entre los nodos n18 y n19, por más que en la otra porción de la red

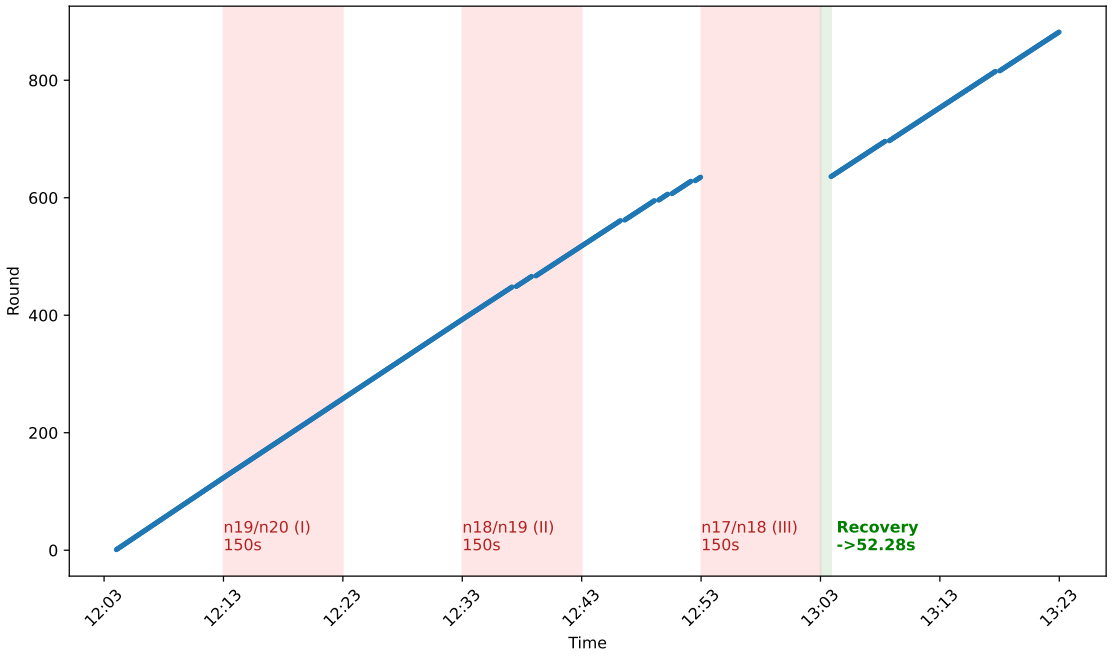


Fig. 3.10: Bloques producidos por la red en todo el experimento tomados del nodo n15, donde el área roja delimita el tiempo donde se aplica el *delay* de 150 s entre los nodos indicados en la figura, mientras que el área verde se corresponde con el tiempo de recuperación de la red para retomar la producción de bloques.

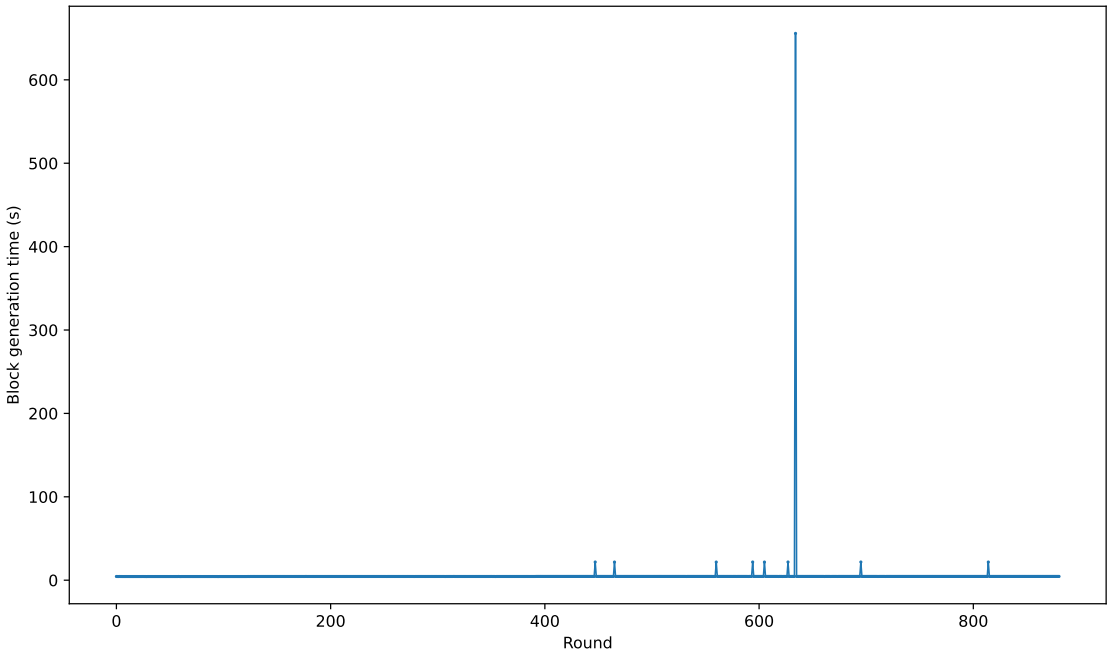


Fig. 3.11: Tiempo para la generación del bloque de cada ronda del experimento.

se sigan generando. En el momento que se levanta este *delay*, el nodo n19 se reconecta y consigue inmediatamente información de todos los bloques que no observó durante el tiempo que estuvo incomunicado del resto de la red, provocando una recta vertical en el gráfico para ese instante.

Con esto podemos confirmar que afectando un 30 % del *stake* de la red podemos frenar la generación de bloques y así impedir su progreso. Sin embargo, esto implica desconectar los *relays* a nivel de la red física, algo no tan trivial para un atacante.

3.5. Red con topología física lineal: *delays* incrementales

Como siguiente paso, queremos encontrar un tiempo de *delay* mínimo para lograr lo mismo que logramos previamente básicamente desconectando a los *relays* del sistema. Sería interesante que, con un esfuerzo pequeño, pudiéramos detener la red o afectarla fuertemente.

Planteamos un nuevo experimento utilizando el mismo sistema de los experimentos anteriores, pero en esta ocasión vamos a aplicar un *delay* incremental sobre el enlace entre n17 y n18: sabemos que necesitamos apartar al menos el 30 % del *stake*, con lo cual si particionamos la red a partir de este enlace, lograríamos el objetivo. El mismo lo planteamos en forma gráfica en la figura 3.14.

El experimento consiste en correr la red por 190 min, aplicando *delays* incrementales.

Sistema:

- **Nodos:** 10 (n1 a n10)
- ***Relays*:** 10 (n11 a n20)
- ***Relays* conectados por nodo:** 1 lógico, 1 físico (20 ms)
- ***Relays* conectados por *relay*:** 4 lógicos, 1 físico (30 ms)

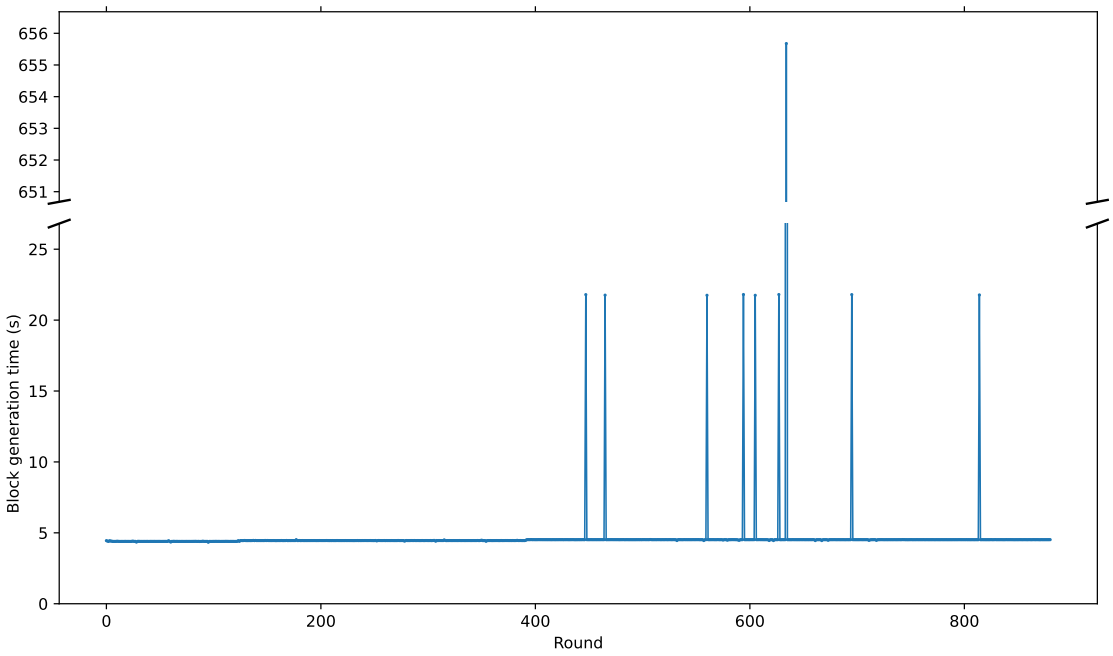


Fig. 3.12: (Detalle) Tiempo para la generación del bloque de cada ronda del experimento.

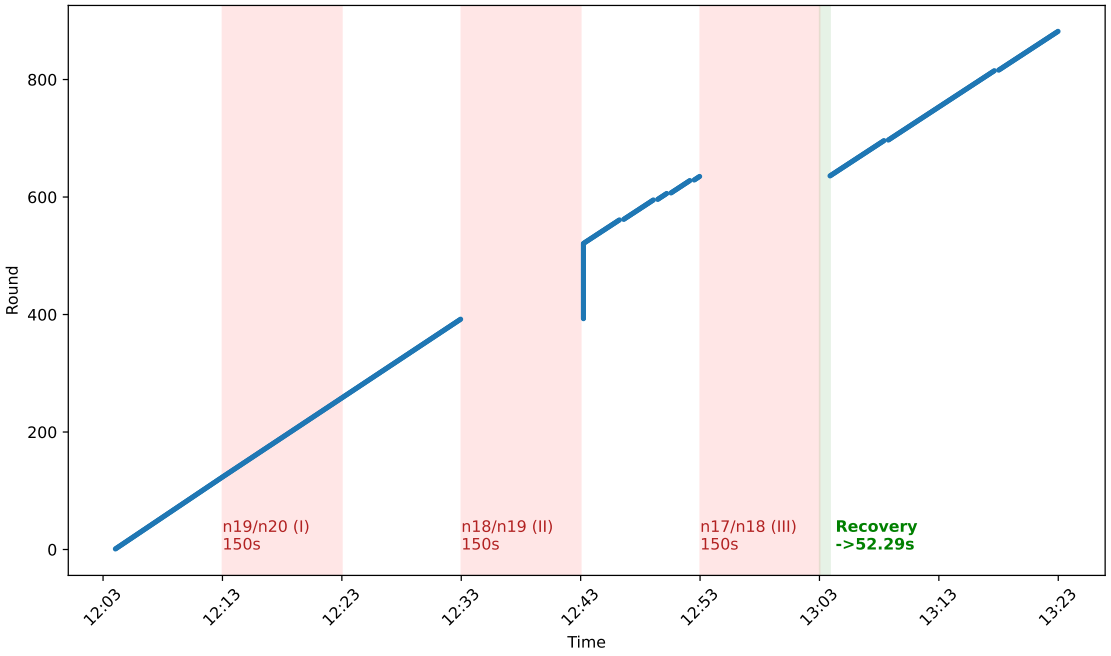


Fig. 3.13: Bloques producidos por la red en todo el experimento observados por el nodo n19, donde el área roja delimita el tiempo donde se aplica el *delay* de 150 s entre los nodos indicados en la figura, mientras que el área verde se corresponde con el tiempo de recuperación de la red para retomar la producción de bloques. En el tiempo 12:43 se observa una recta que se corresponde con el nodo n19 obteniendo toda la información que no pudo recibir mientras se encontraba afectado por el *delay*, luego de su proceso de reconexión y recuperación.

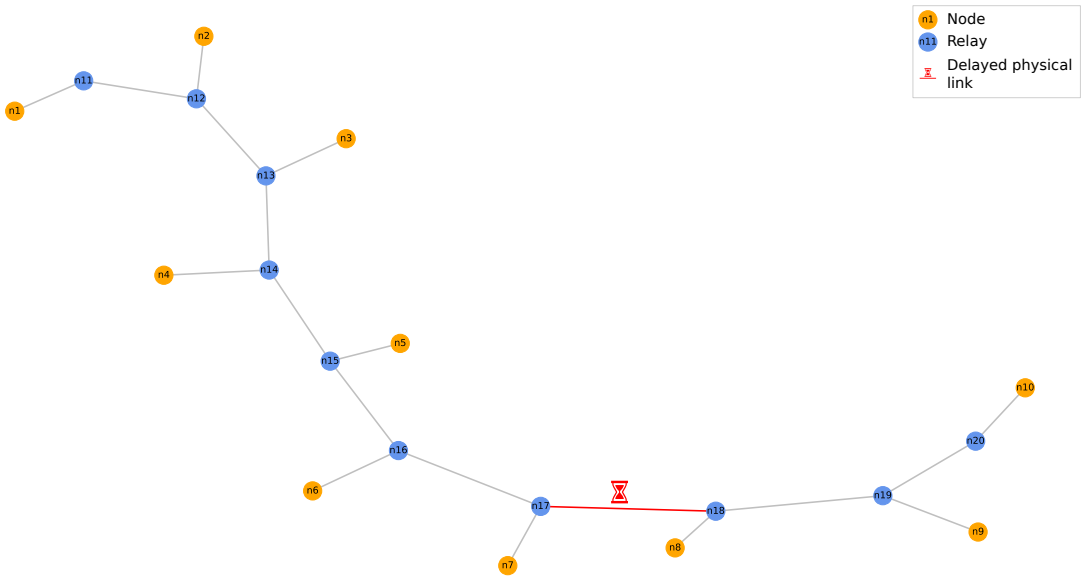


Fig. 3.14: Topología física, se muestran los enlaces afectados en la red, agregando *delays* en el enlace entre los nodos n17 y n18.

- ***Stake*:** homogéneo
- **Tiempo de ejecución:** 190 min
- **Servidor:** local (1 servidor)
- ***Delays* aplicados (según rango de minutos) en n17-n18:**
  - 0 min a 10 min: 30 ms
  - 10 min a 30 min: 1 s
  - 30 min a 50 min: 2 s
  - 50 min a 70 min: 3 s
  - 70 min a 90 min: 4 s
  - 90 min a 110 min: 5 s
  - 110 min a 130 min: 6 s
  - 130 min a 150 min: 7 s
  - 150 min a 170 min: 8 s
  - 170 min a 190 min: 30 ms

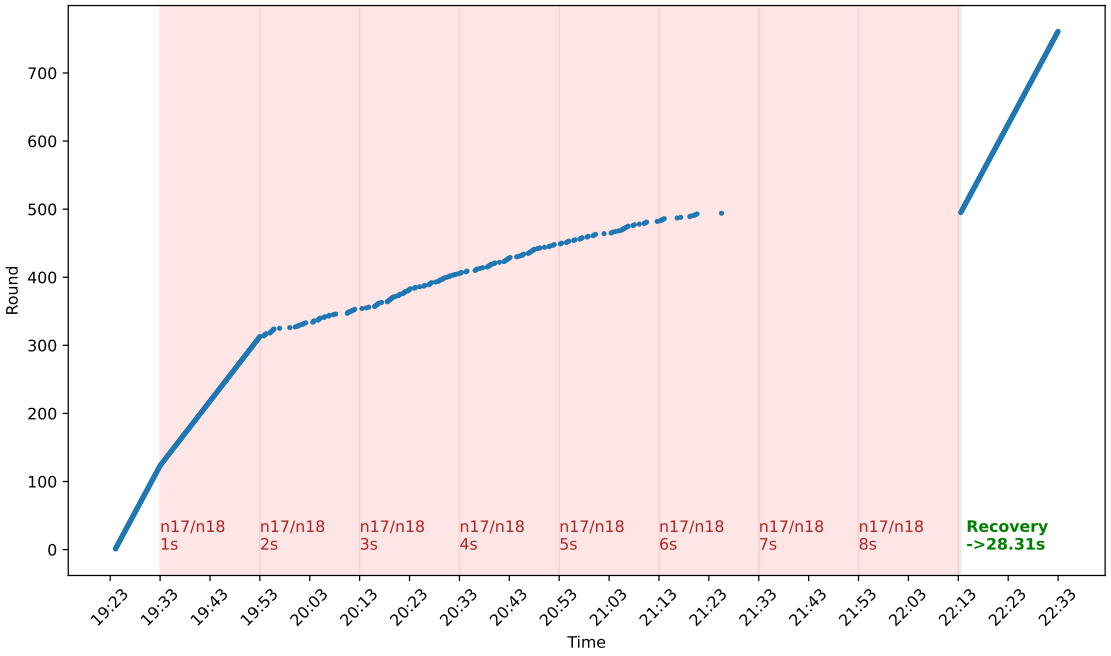


Fig. 3.15: Bloques producidos por la red en todo el experimento tomados del nodo n15, donde el área roja delimita el tiempo donde se aplica el *delay* entre los nodos indicados en la figura, mientras que el área verde se corresponde con el tiempo de recuperación de la red para retomar la producción de bloques.

En las figuras 3.15, 3.16 y 3.17 vemos que la red comienza generando bloques normalmente cada 4,3 s. Al aplicar el *delay* de 1 s, este tiempo aumenta a 6,5 s (observable en la tercera figura). Sin embargo, se siguen generando bloques sin interrupción.

Con *delay* de 2 s la red comienza a degradarse y notamos un incremento notable en los tiempos de generación de bloque, teniendo picos de más de 100 s. Esto implica un aumento de más de 26 veces el tiempo de generación normal. El impacto es muy importante para un *delay* no tan grande como son 2 s en el enlace. A partir de este punto, la red continúa avanzando, aunque lentamente, hasta llegar al *delay* de 6 s. Con este valor vemos que, si bien no es instantáneo, la red no logra avanzar más luego de unos minutos de que

fue aplicado. Este comportamiento se mantiene al incrementar el *delay* a 7 s y 8 s. Recién luego de que se levanta el *delay* puede retomar su normal funcionamiento la red (tomándole alrededor de 28 s lograrlo, siendo el primer bloque generado en más de 47 min). En este caso, el tiempo de recuperación es notablemente menor que en los casos anteriores. Esto podría

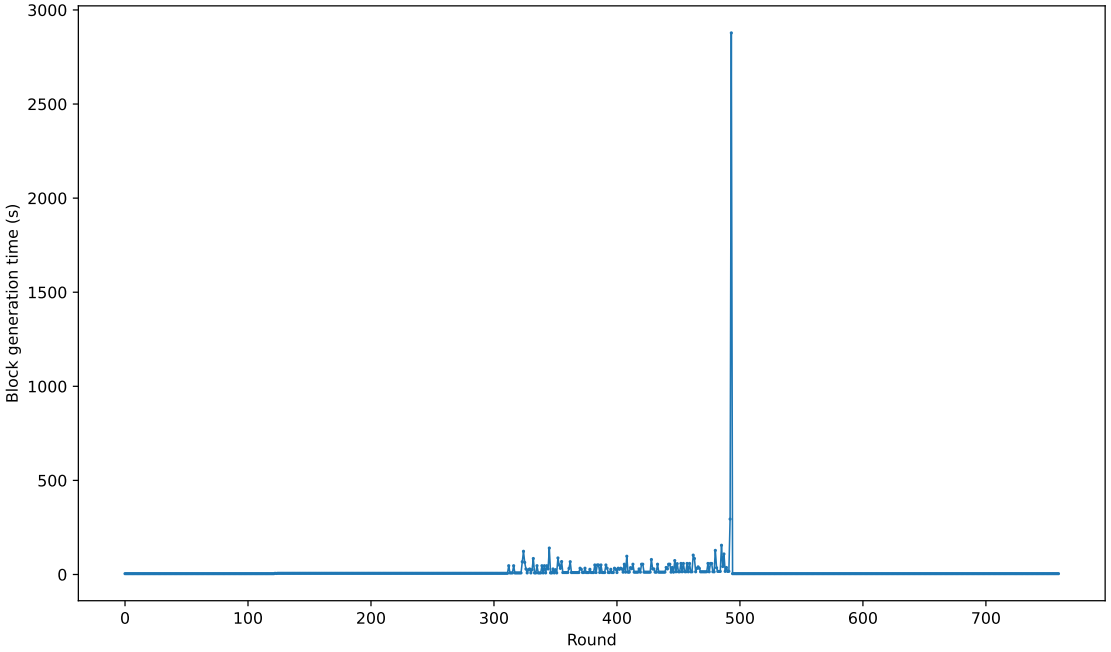


Fig. 3.16: Tiempo para la generación del bloque de cada ronda del experimento.

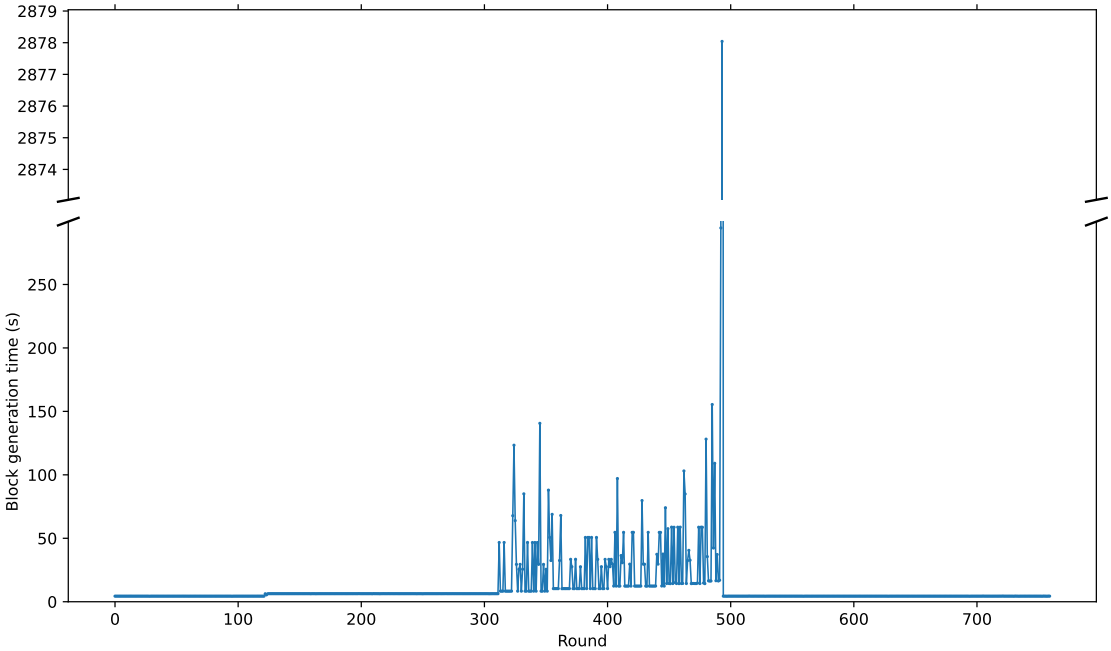


Fig. 3.17: (Detalle) Tiempo para la generación del bloque de cada ronda del experimento.

deberse a que si bien el *delay* introducido en el sistema logra alterar su funcionamiento, no aisla completamente a los nodos afectados: los demás nodos conocen de su existencia y reciben mensajes demorados debido al *delay*, pero no parecen desconectados como en el caso anterior.

Estos resultados nos otorgan una primer indicio para considerar: a partir de un *delay* de 2 s podemos comprometer el sistema agregando una demora muy importante en su avance, pudiendo incluso detenerlo con uno de 6 s en el lugar indicado.

Todos estos casos responden a una topología física que resulta fácil de atacar: al estar dispuestos como una línea, alcanza con demorar un solo enlace para poder dividir la misma en dos y separar los *stakes*. Avanzamos nuestros experimentos incrementando la dificultad al unir ambos extremos de la línea para conformar un anillo de *relays*.

3.6. Red con topología física de anillo

Con esta nueva topología agregamos mayor dificultad para un atacante, dado que ahora existen dos vías de comunicación para los *relays*: agregamos un enlace entre el nodo n20 y el n11, logrando una topología física de anillo para la red, representada en la imagen 3.18.

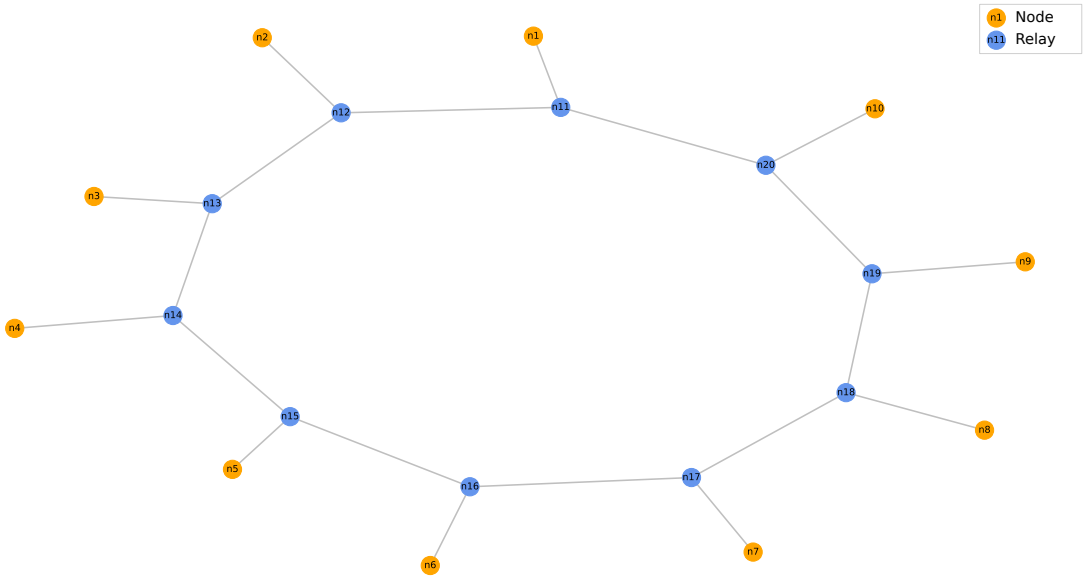


Fig. 3.18: Topología de la red física.

En cuanto a la red lógica, se mantiene de la misma manera que en los experimentos anteriores, dado que no nos interesa alterarla: justamente queremos ver que dada una configuración lógica, podemos modificar la topología sobre la que corre para lograr que deje de funcionar.

Con esta nueva arquitectura no nos alcanza con alterar un solo enlace para afectar el progreso, ahora debemos afectar dos enlaces para poder aislar el 30 % del *stake* y así lograrlo.

Para esto afectaremos los enlaces entre n17 y n18, y entre n20 y n11, dejando a los nodos n18, n19 y n20 aislados de los demás (resultando en lo planteado en el gráfico 3.19). Al igual que en el experimento anterior, agregamos *delays* incrementales, aumentando en

1 s el mismo en cada iteración. Buscamos nuevamente cuál es el valor de *delay* necesario para afectar la red.

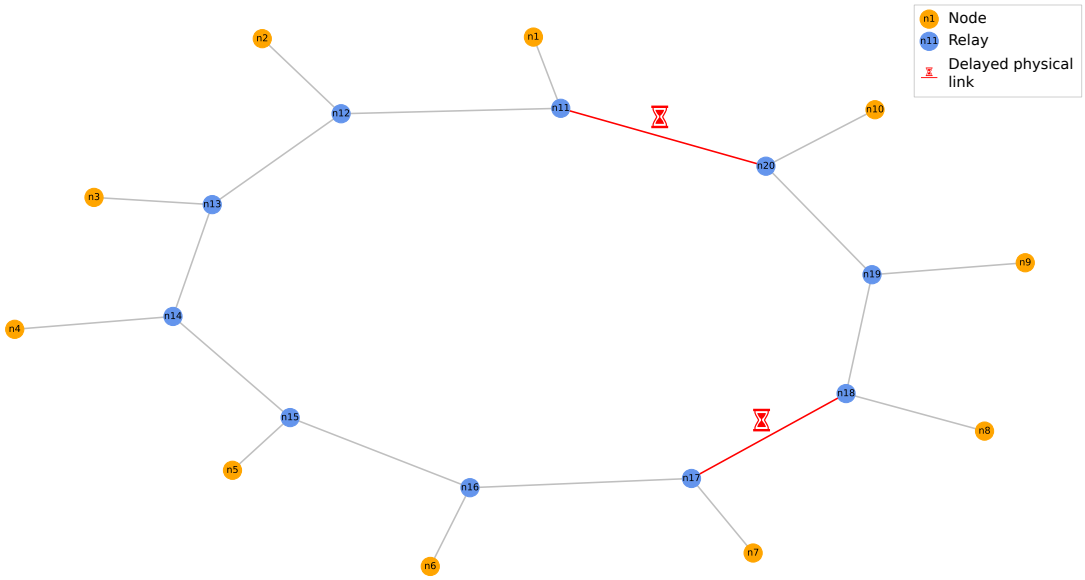


Fig. 3.19: Topología física, se muestran los enlaces afectados en la red, agregando *delays* en el enlace entre los nodos n11 y n20, y entre los nodos n17 y n18.

Sistema:

- **Nodos:** 10 (n1 a n10)
- ***Relays*:** 10 (n11 a n20)
- ***Relays* conectados por nodo:** 1 lógico, 1 físico (20 ms)
- ***Relays* conectados por *relay*:** 4 lógicos, 1 físico (30 ms)
- ***Stake*:** homogéneo
- **Tiempo de ejecución:** 190 min
- **Servidor:** local (1 servidor)
- ***Delays* aplicados (según rango de minutos) en n17-n18 y n20-n11:**
  - 0 min a 10 min: 30 ms
  - 10 min a 30 min: 1 s
  - 30 min a 50 min: 2 s
  - 50 min a 70 min: 3 s
  - 70 min a 90 min: 4 s
  - 90 min a 110 min: 5 s
  - 110 min a 130 min: 6 s
  - 130 min a 150 min: 7 s
  - 150 min a 170 min: 8 s
  - 170 min a 190 min: 30 ms

El sistema responde a la misma configuración que el experimento de *delays* anterior, con la motivación de analizar qué diferencias encontramos al modificar la topología física.

En base a los resultados observados en las figuras 3.20, 3.21 y 3.22, la red parece más estable con los primeros *delays* en comparación con lo obtenido en la red con topología física lineal: con el *delay* de 1 s el tiempo de ronda pasa de 4,3 s a 6,2 s. Con 2 s se incrementa

a 8,2 s, un poco menos del doble de lo normal, pero notamos que la red se mantiene

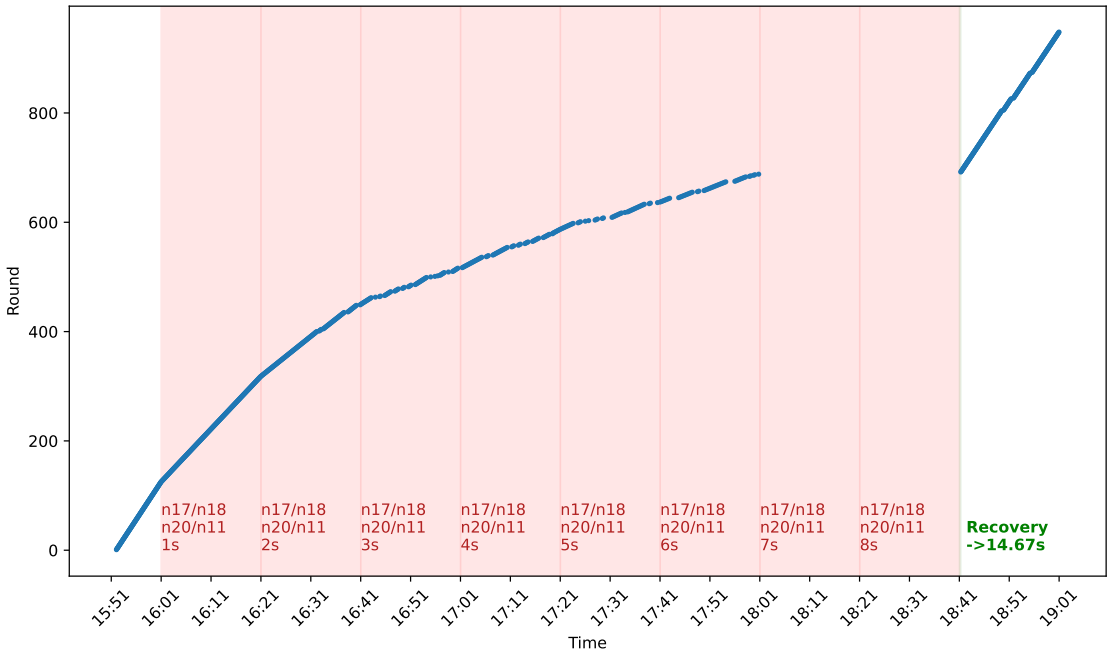


Fig. 3.20: Bloques producidos por la red en todo el experimento tomados del nodo n15, donde el área roja delimita el tiempo donde se aplica el *delay* entre los nodos indicados en la figura, mientras que el área verde se corresponde con el tiempo de recuperación de la red para retomar la producción de bloques.

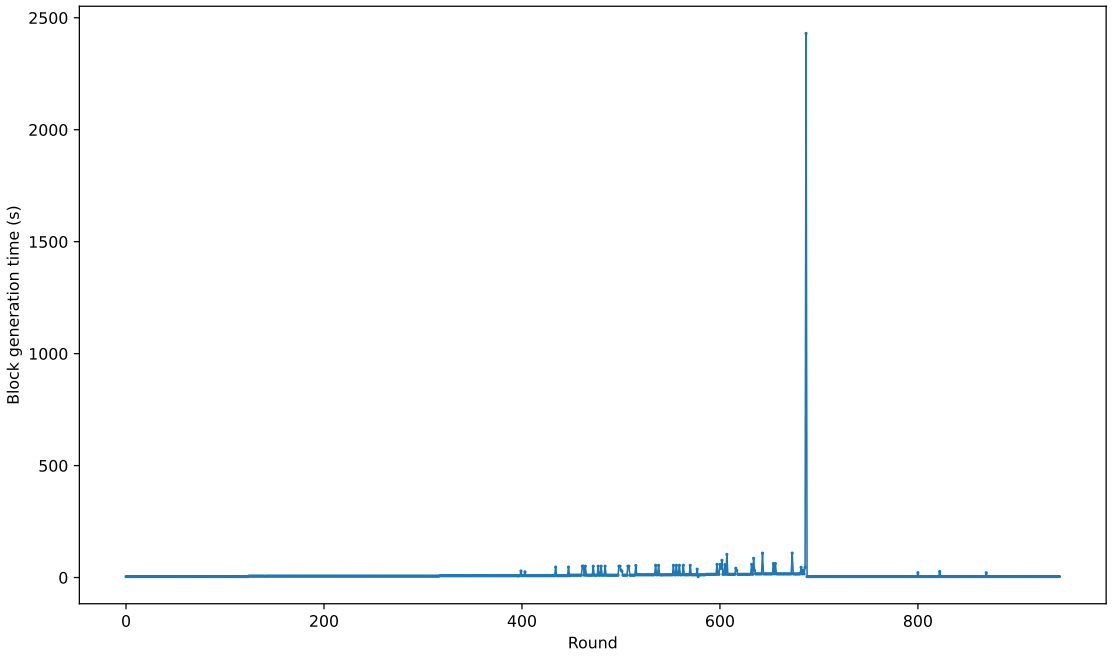


Fig. 3.21: Tiempo para la generación del bloque de cada ronda del experimento.

constante en la generación de bloques, a diferencia de la red lineal. A partir de los 3 s notamos que se degrada la generación de bloques, teniendo demoras desde 10 s hasta 54 s. Con el incremento del *delay* se observan algunos otros picos en los tiempos de generación de bloque, hasta que a partir de los 7 s la red deja de producir bloques.

Al restablecer el enlace a los 30 ms iniciales, la red recupera su ritmo normal de avance luego de un período de solo 14 s, una muy buena marca considerando el tiempo que se mantiene inactiva. Pero por lo visto previamente, la recuperación ante un *delay* relativamente bajo es mucho más rápida que en el caso de uno muy grande (como el de 150 s).

Al aumentar la complejidad de la topología física, también incrementamos la dificultad del ataque: ahora necesitamos atacar dos enlaces en vez de uno, y el tiempo de *delay* necesario crece a 7 s para cada uno. A su vez, el impacto de los *delays* menores no parece tan grande como en el caso anterior, aunque igualmente degrada la generación de bloques.

Continuamos incrementando la dificultad considerando ahora una red más elaborada, con mayor cantidad de conexiones entre los *relays*.

3.7. Red de *relays* en clique física

Como siguiente paso, aumentamos la complejidad de la red generando una cliqué física entre los *relays* del sistema, siendo graficado en la imagen 3.23. De esta manera, cada *relay* tiene una conexión física directa a todos los demás *relays*, aunque mantenemos las conexiones lógicas como estaban (es decir, siguen siendo 4 por *relay*, como se muestra en la imagen 3.24). Si bien no resulta realista una topología física como la planteada, en vistas de que requeriría una interfaz de red por cada enlace del *relay*, lo planteamos como el caso más elaborado posible de la conectividad entre estos actores.

En este caso, la topología de la red lógica termina siendo un subconjunto de los enlaces

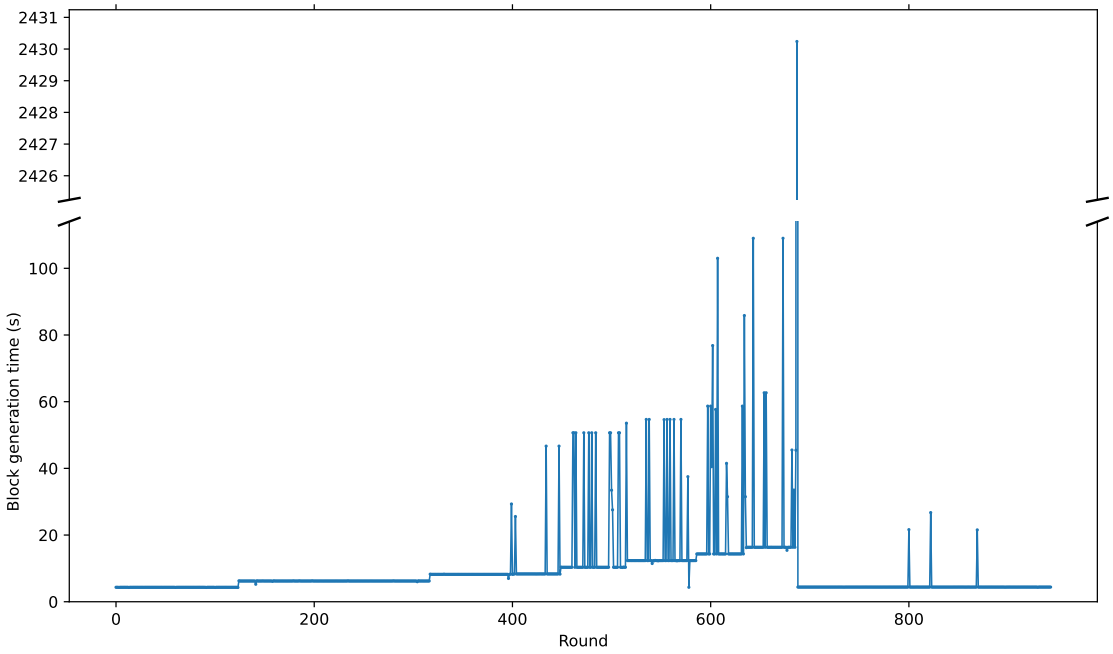


Fig. 3.22: (Detalle) Tiempo para la generación del bloque de cada ronda del experimento.

de la topología física. Por este motivo, para poder realizar nuestro ataque a los *relays*, vamos a necesitar agregarle latencia a todos los enlaces que parten de cada uno de ellos. En vistas de que hay nueve enlaces que conectan a cada *relay* con los demás, necesitamos agregarle *delay* a cada uno de ellos. En este caso solo vamos a atacar esos enlaces, sin incluir los enlaces hacia los nodos que están conectados a cada *relay* (caso que veremos posteriormente).

Vamos a tomar la misma metodología aplicada para los experimentos anteriores y agregamos demoras incrementales en todos estos enlaces para así encontrar en qué valor

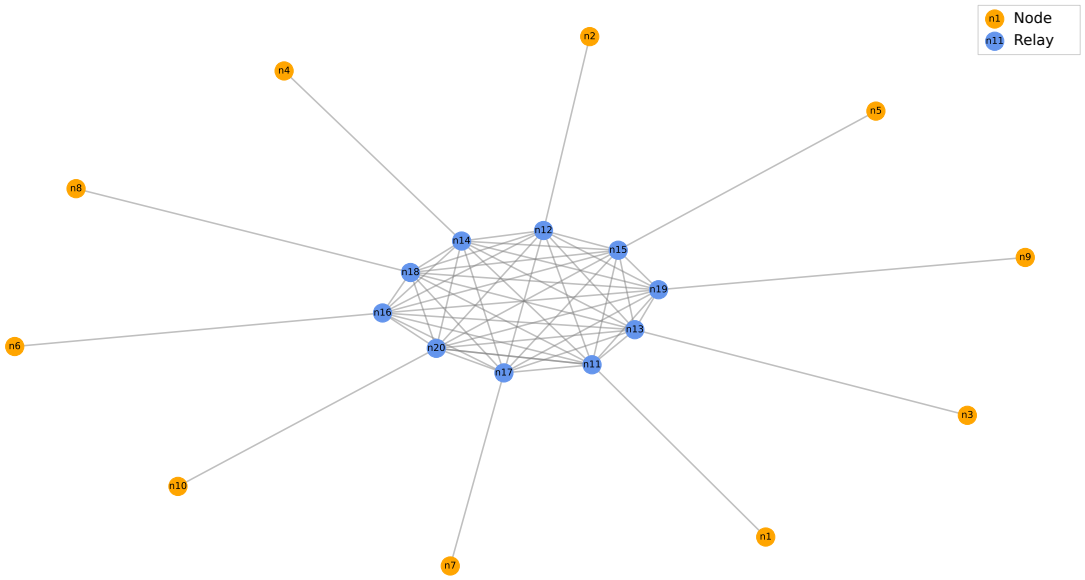


Fig. 3.23: Topología de la red física.

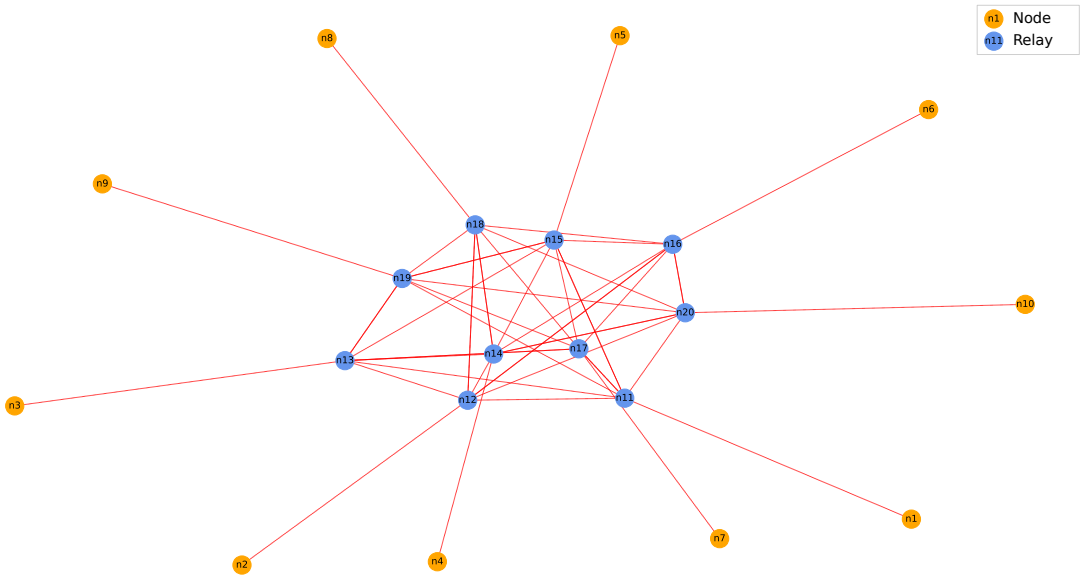


Fig. 3.24: Topología de la red lógica.

el sistema se degrada o deja de producir bloques. Atacamos los *relays* n18, n19 y n20, intentando aislar así el 30 % del *stake* participante del consenso, resultando en el ataque mostrado en la imagen 3.25.

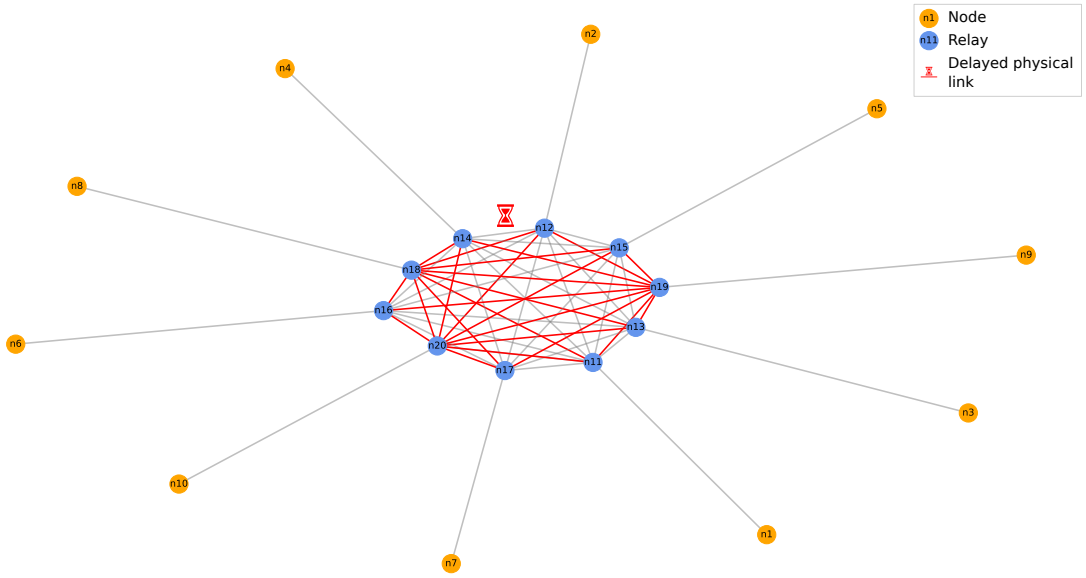


Fig. 3.25: Topología física, se muestran los enlaces afectados en la red, agregando *delays* en todos los enlaces de los *relays* n18, n19 y n20 hacia los demás *relays*.

Sistema:

- **Nodos:** 10 (n1 a n10)
- ***Relays*:** 10 (n11 a n20)
- ***Relays* conectados por nodo:** 1 lógico, 1 físico (20 ms)
- ***Relays* conectados por *relay*:** 4 lógicos, 9 físicos (30 ms)
- ***Stake*:** homogéneo
- **Tiempo de ejecución:** 190 min
- **Servidor:** local (1 servidor)
- ***Delays* aplicados (según rango de minutos) en n18, n19 y n20 hacia todos los *relays*:**
  - 0 min a 10 min: 30 ms
  - 10 min a 30 min: 1 s
  - 30 min a 50 min: 2 s
  - 50 min a 70 min: 3 s
  - 70 min a 90 min: 4 s
  - 90 min a 110 min: 5 s
  - 110 min a 130 min: 6 s
  - 130 min a 150 min: 7 s
  - 150 min a 170 min: 8 s
  - 170 min a 190 min: 30 ms

Los resultados observados en las figuras 3.26, 3.27 y 3.28 se condicen con lo observado en los casos previos: con el primer *delay* de 1 s aumenta el tiempo de producción de bloques, pero se mantiene constante. Luego, a partir de los 2 s se empieza a degradar este

ritmo, produciéndose con mayor demora y algunos leves sobresaltos. Esto se mantiene

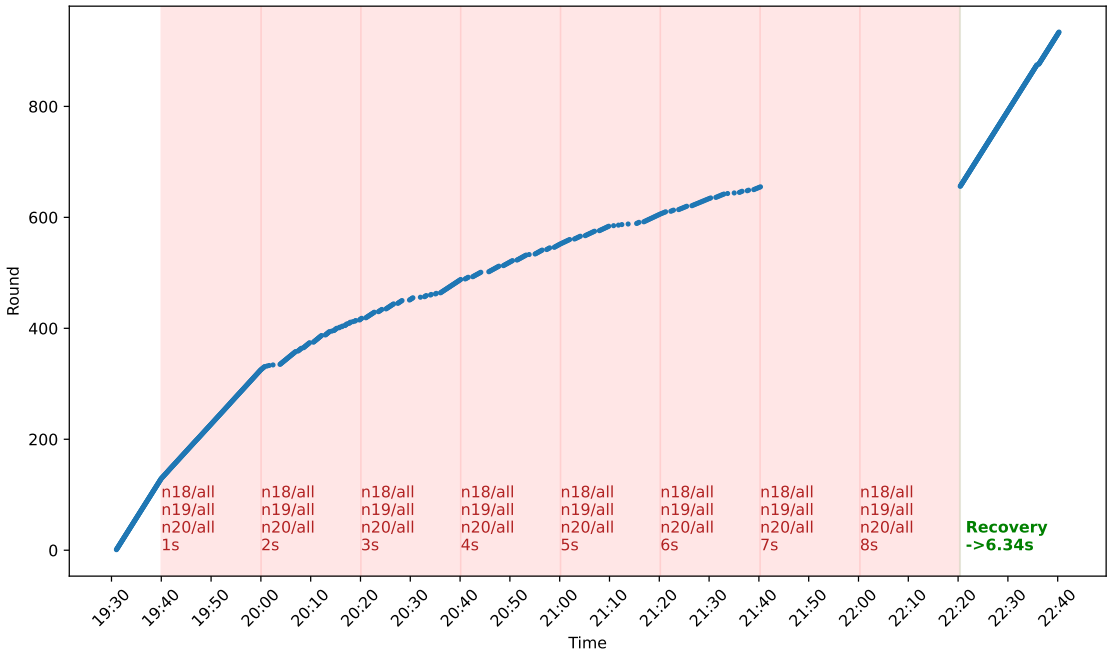


Fig. 3.26: Bloques producidos por la red en todo el experimento tomados del nodo n15, donde el área roja delimita el tiempo donde se aplica el *delay* entre los nodos indicados en la figura, mientras que el área verde se corresponde con el tiempo de recuperación de la red para retomar la producción de bloques.

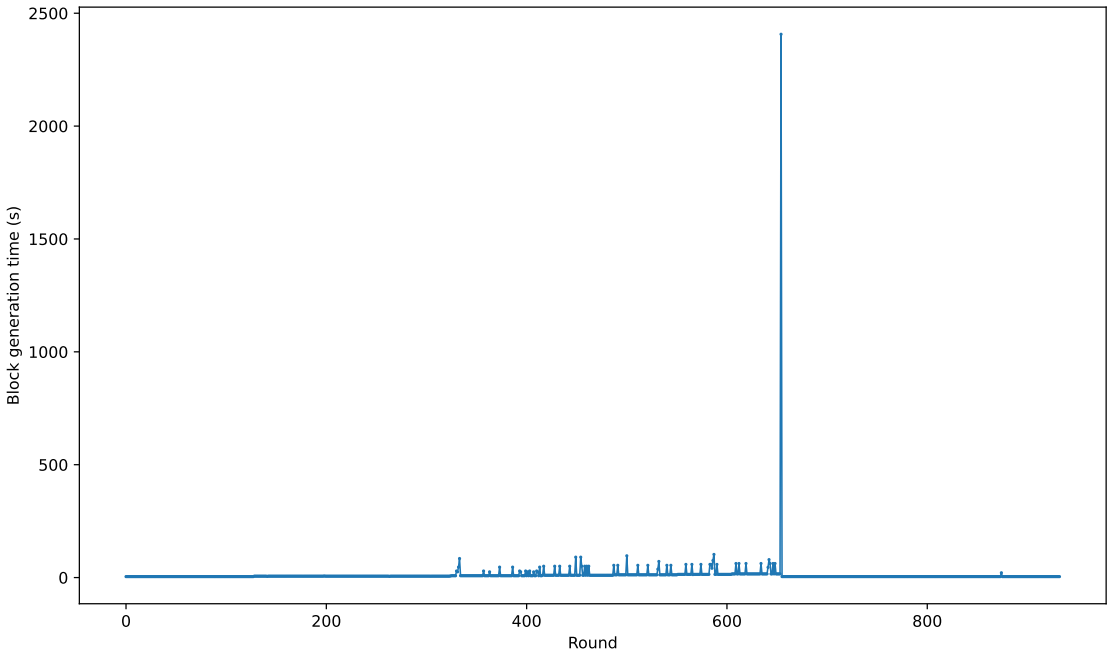


Fig. 3.27: Tiempo para la generación del bloque de cada ronda del experimento.

hasta los 7 s, momento en el cual la red no puede avanzar más y no se generan bloques, deteniéndose. Finalmente, cuando se restablece el tiempo de latencia inicial, el sistema retoma la producción de bloques en solo 6 s.

Dado que son nueve enlaces los atacados por *relay*, terminamos atacando 27 enlaces con un *delay* de 7 s, lo cual requiere un esfuerzo sumamente superior a lo que veíamos previamente. Esto no es un dato menor, dado que para que un ataque sea factible, queremos que la cantidad de enlaces y *delays* sean los más bajos posibles.

Con esta motivación, exploramos qué sucede si además de atacar todos estos enlaces de cada uno de los *relays* también incluimos las conexiones a los nodos que están asociados a ellos.

3.8. Red de *relays* en clique física: *delays* incluyendo nodos

Uno de los problemas del experimento anterior es que solo afectamos los enlaces que conectan entre ellos a los *relays*, sin aprovechar el hecho de que los participantes más importante del consenso son los nodos conectados a ellos. Cada mensaje que recibe un *relay* durante el avance de la red debe ser enviado al nodo, el nodo procesa el contenido, y potencialmente envía mensajes en base a lo recibido hacia la red. En este trayecto, se demora al menos dos veces el tiempo de latencia de la conexión entre el nodo y el *relay* entre el recibir y el enviar el mensaje. Luego, si nosotros atacáramos también este enlace, sumaríamos rápidamente una demora importante en la toma de decisiones debido a cómo se construye la arquitectura de la red.

El sistema para el experimento es el mismo que el caso anterior, pero ahora vamos a agregar *delay* también entre los *relays* atacados (n18, n19 y n20) y sus nodos conectados (n8, n9 y n10 respectivamente), siendo graficado en la imagen 3.29.

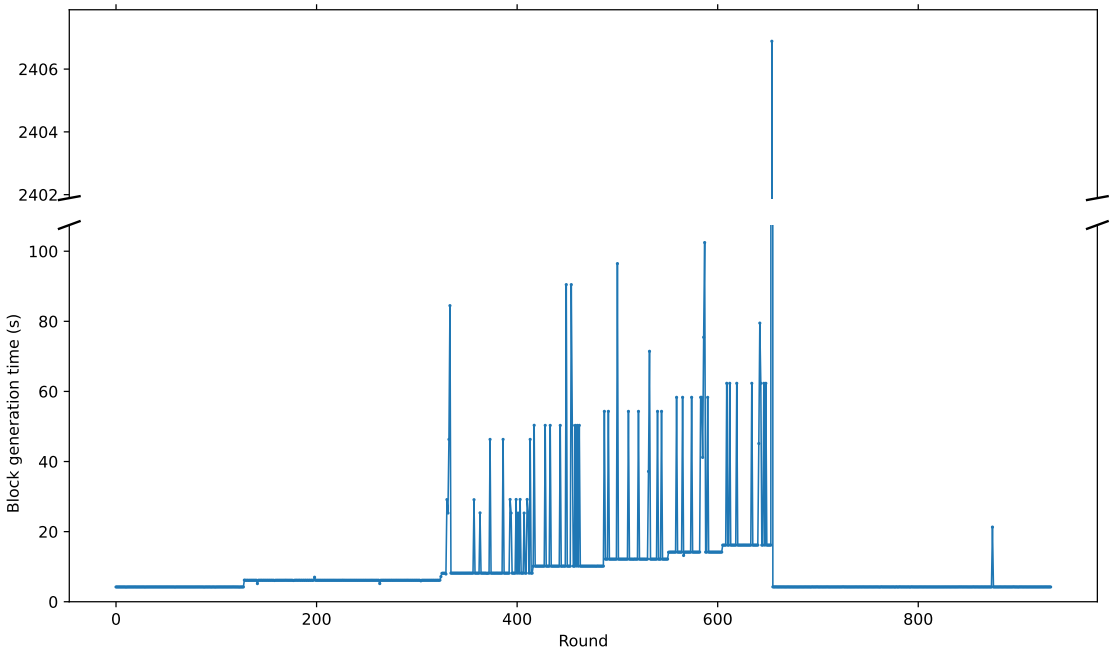


Fig. 3.28: (Detalle) Tiempo para la generación del bloque de cada ronda del experimento.

Sistema:

- **Nodos:** 10 (n1 a n10)
- ***Relays*:** 10 (n11 a n20)
- ***Relays* conectados por nodo:** 1 lógico, 1 físico (20 ms)
- ***Relays* conectados por *relay*:** 4 lógicos, 9 físicos (30 ms)
- ***Stake*:** homogéneo
- **Tiempo de ejecución:** 190 min
- **Servidor:** local (1 servidor)
- ***Delays* aplicados (según rango de minutos) en n18, n19 y n20 hacia todos los *relays* y sus respectivos nodos (n8, n9 y n10):**
  - 0 min a 10 min: 30 ms
  - 10 min a 30 min: 1 s
  - 30 min a 50 min: 2 s
  - 50 min a 70 min: 3 s
  - 70 min a 90 min: 4 s
  - 90 min a 110 min: 5 s
  - 110 min a 130 min: 6 s
  - 130 min a 150 min: 7 s
  - 150 min a 170 min: 8 s
  - 170 min a 190 min: 30 ms

Planteamos los resultados para el experimento en las figuras 3.30, 3.31 y 3.32. Para el primer *delay* de 1 s el impacto ya es mayor que en los casos anteriores: el tiempo de ronda pasa de 4,3 s a 8 s, casi el doble. Luego, con 2 s y 3 s notamos una degradación importante en los tiempos de generación, llegando a tener picos entre 50 s a 100 s de demora. Pero el mayor logro pasa a ser a partir de los 4 s de *delay*, momento en el que la red no puede

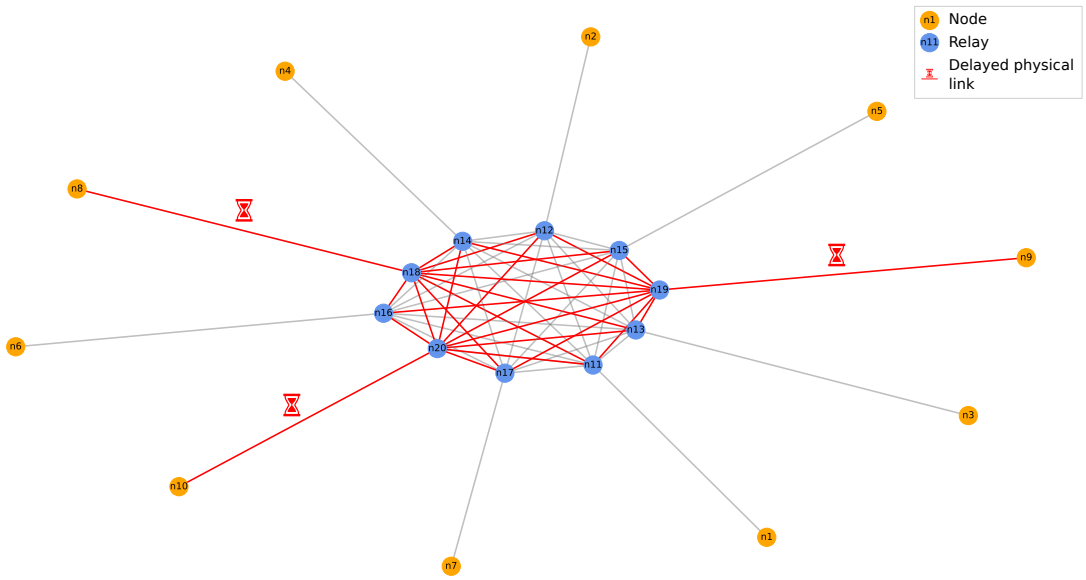


Fig. 3.29: Topología física, se muestran los enlaces afectados en la red, agregando *delays* en todos los enlaces de los *relays* n18, n19 y n20 hacia los demás *relays* y sus propios nodos asociados.

avanzar más y se detiene la producción de bloques hasta que se restablecen los enlaces

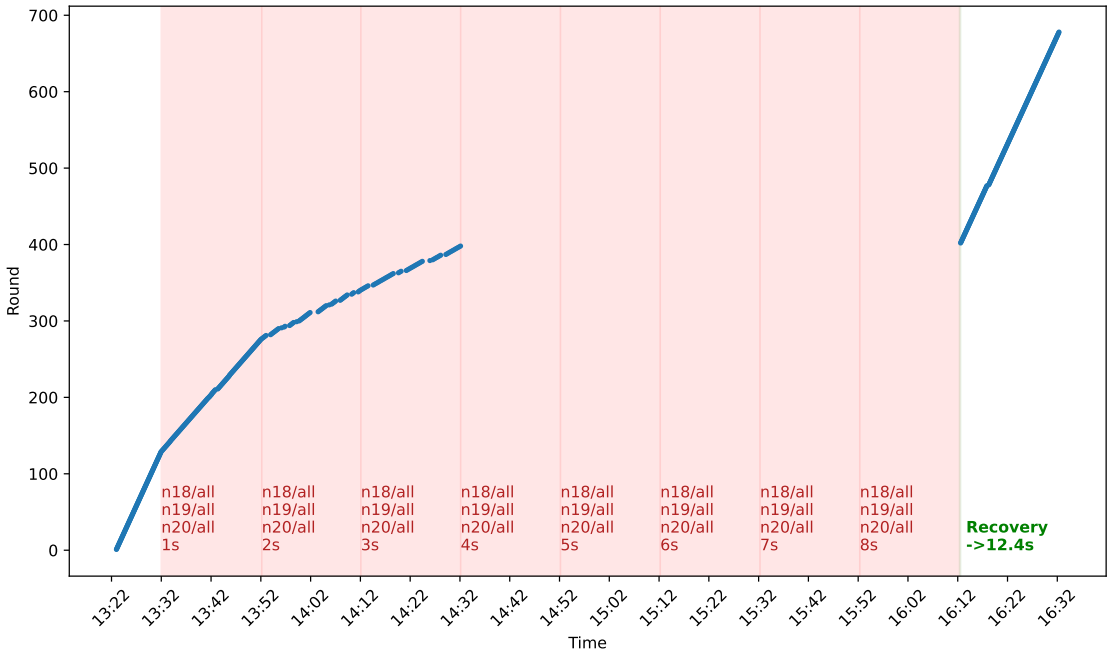


Fig. 3.30: Bloques producidos por la red en todo el experimento tomados del nodo n15, donde el área roja delimita el tiempo donde se aplica el *delay* entre los nodos indicados en la figura, mientras que el área verde se corresponde con el tiempo de recuperación de la red para retomar la producción de bloques.

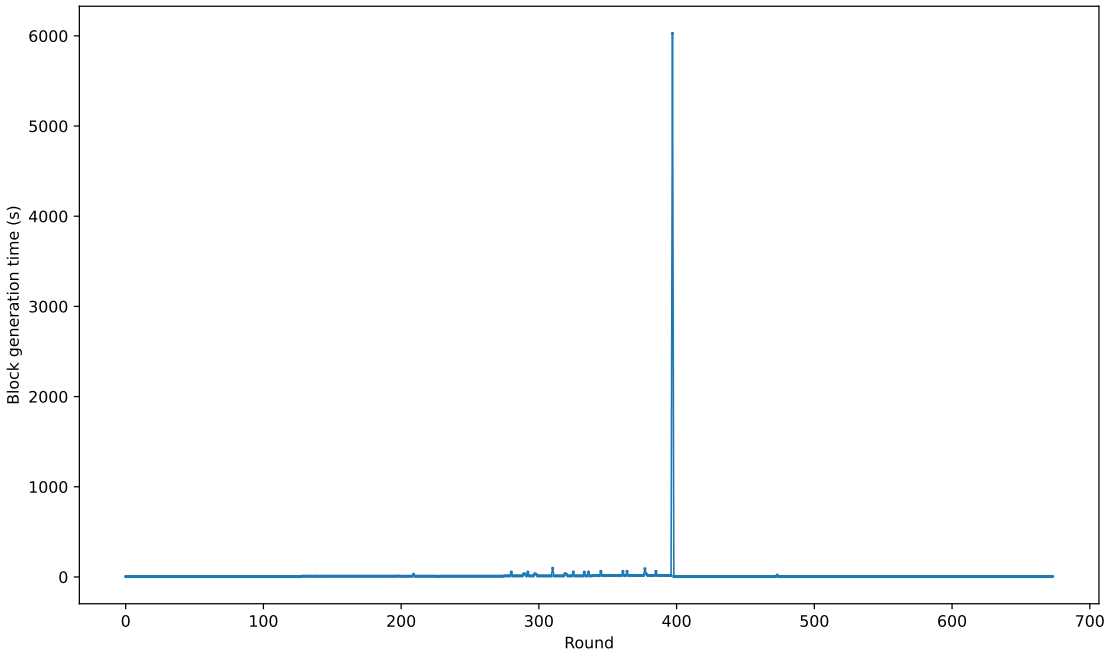


Fig. 3.31: Tiempo para la generación del bloque de cada ronda del experimento.

(cuyo tiempo de recuperación en este caso es de 12 s).

Esta mejora en el tiempo necesario para detener el sistema es sumamente significativa: necesitamos casi la mitad del esfuerzo que en el caso anterior donde atacamos solo las conexiones entre *relays*. Este punto es clave, ya que conociendo la arquitectura de la red `Algorand`, sabemos que los enlaces entre nodos y *relays* son un punto crítico para el envío de mensajes. Si bien luego los *relays* son los encargados de propagarlos y si fueran comprometidos podría también resultar sumamente insalubre para la red, estos solo envían los mensajes recibidos a un subconjunto de pares en el caso de que fueran nuevos o no fueran recibidos de parte de los mismos (como parte de su mecanismo de *flooding*).

Con estos resultados ahora nos preguntamos si es necesario atacar todos los enlaces de los *relays*, o si es suficiente con atacar la conexión entre cada nodo y su *relay* correspondiente.

3.9. Red de *relays* en clique física: *delays* solo en nodos

Según lo visto en el experimento anterior, agregar *delay* en la conexión entre un nodo y el *relay* al que se encuentra conectado resulta muy efectivo para afectar la red. Sin embargo, en el caso visto son muchos los enlaces totales a afectar para lograrlo. Si solo atacáramos los enlaces nodo/*relay*, con agregar una latencia considerable entre tres de ellos alcanzaría para detener la red, reduciendo drásticamente el esfuerzo necesario.

Planteamos el mismo sistema que el caso anterior (mantiendo la topología física y lógica), ahora solo afectando la conexión entre los *relays* n18, n19 y n20 con sus nodos asociados n8, n9 y n10, planteándolo en forma gráfica en la figura 3.33.

Sistema:

- **Nodos:** 10 (n1 a n10)

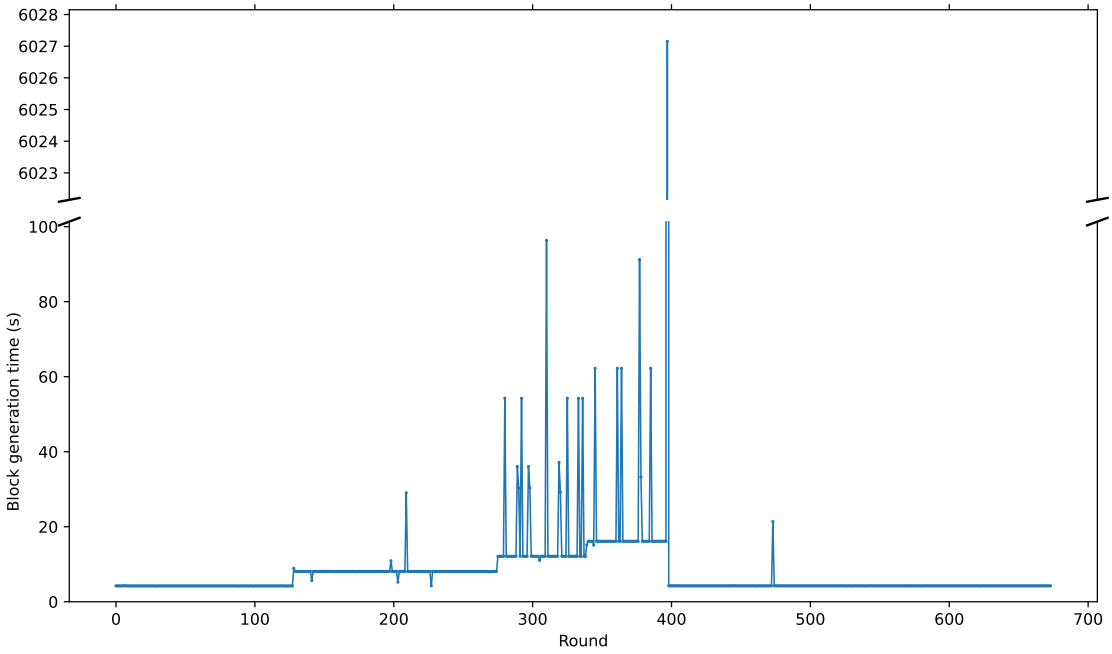


Fig. 3.32: (Detalle) Tiempo para la generación del bloque de cada ronda del experimento.

- ***Relays*:** 10 (n11 a n20)
- ***Relays* conectados por nodo:** 1 lógico, 1 físico (20 ms)
- ***Relays* conectados por *relay*:** 4 lógicos, 9 físicos (30 ms)
- ***Stake*:** homogéneo
- **Tiempo de ejecución:** 190 min
- **Servidor:** local (1 servidor)
- ***Delays* aplicados (según rango de minutos) en n18, n19 y n20 hacia sus respectivos nodos (n8, n9 y n10):**
  - 0 min a 10 min: 30 ms
  - 10 min a 30 min: 1 s
  - 30 min a 50 min: 2 s
  - 50 min a 70 min: 3 s
  - 70 min a 90 min: 4 s
  - 90 min a 110 min: 5 s
  - 110 min a 130 min: 6 s
  - 130 min a 150 min: 7 s
  - 150 min a 170 min: 8 s
  - 170 min a 190 min: 30 ms

El resultado de esta experiencia se observa en las figuras 3.34, 3.35 y 3.36, y resulta similar al de los *relays* en clique sin afectar las conexiones a los nodos: vemos que con el primer *delay* de 1 s aumenta el tiempo de ronda, pero se mantiene constante la producción de bloques. Luego, a partir de los 2 s, la producción se degrada y observamos picos de hasta 62 s mientras se agranda el tiempo de *delay*. A partir de los 7 s la red se detiene y le resulta imposible continuar generando bloques debido a la demora en los mensajes de los nodos afectados.

Si bien el *delay* agregado es la misma que en la otra experiencia, en este caso solo lo

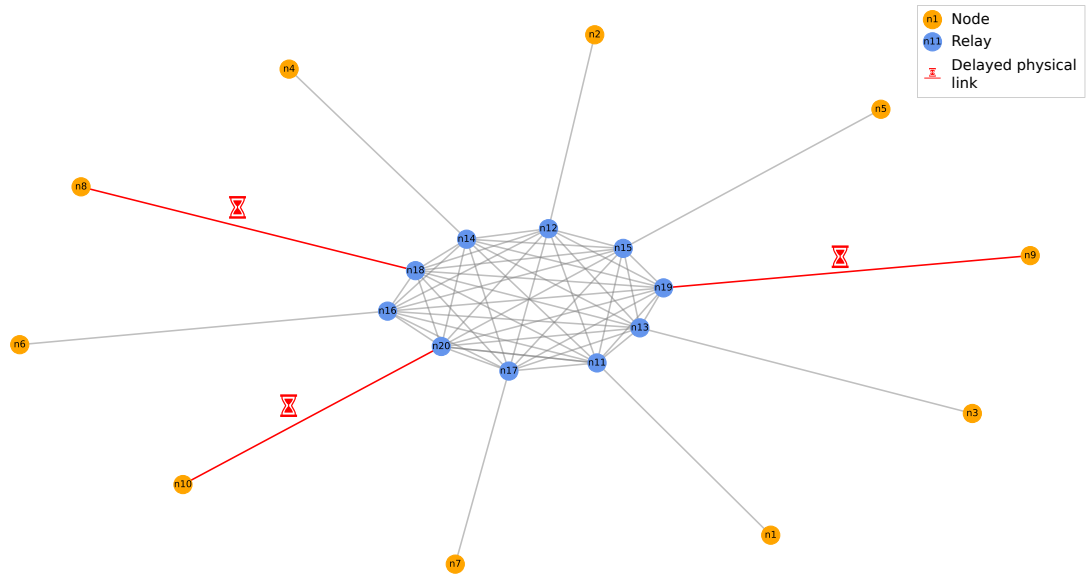


Fig. 3.33: Topología física, se muestran los enlaces afectados en la red, agregando *delays* en todos los enlaces de los *relays* n18, n19 y n20 hacia sus propios nodos asociados.

estamos haciendo sobre tres enlaces de la red, lo cual es enormemente menos complejo que

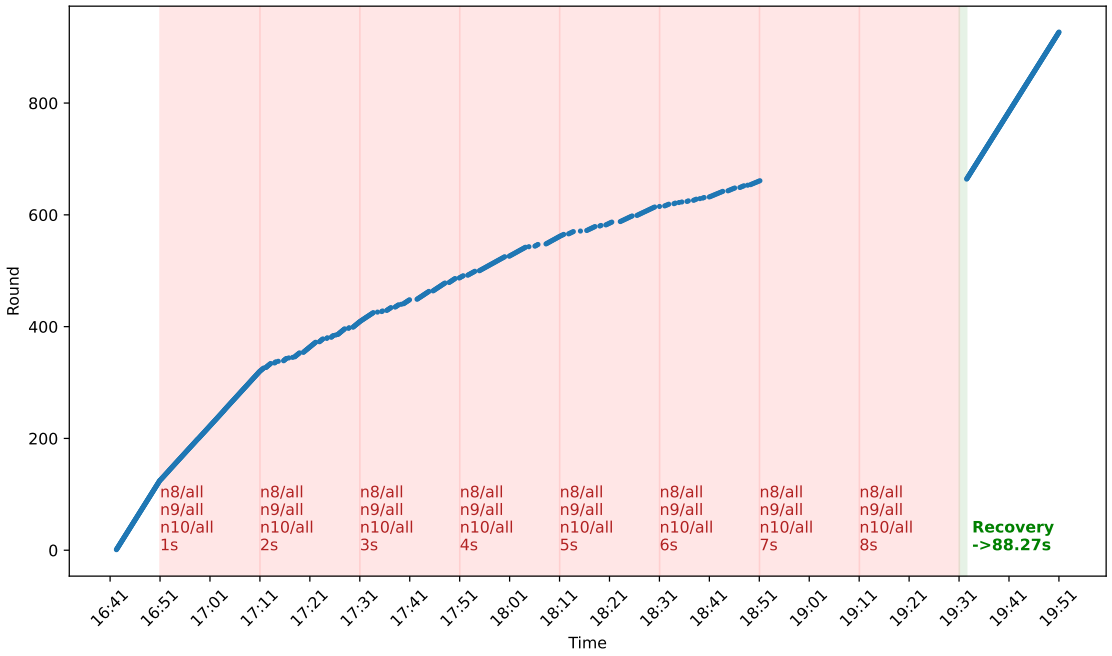


Fig. 3.34: Bloques producidos por la red en todo el experimento tomados del nodo n15, donde el área roja delimita el tiempo donde se aplica el *delay* entre los nodos indicados en la figura, mientras que el área verde se corresponde con el tiempo de recuperación de la red para retomar la producción de bloques.

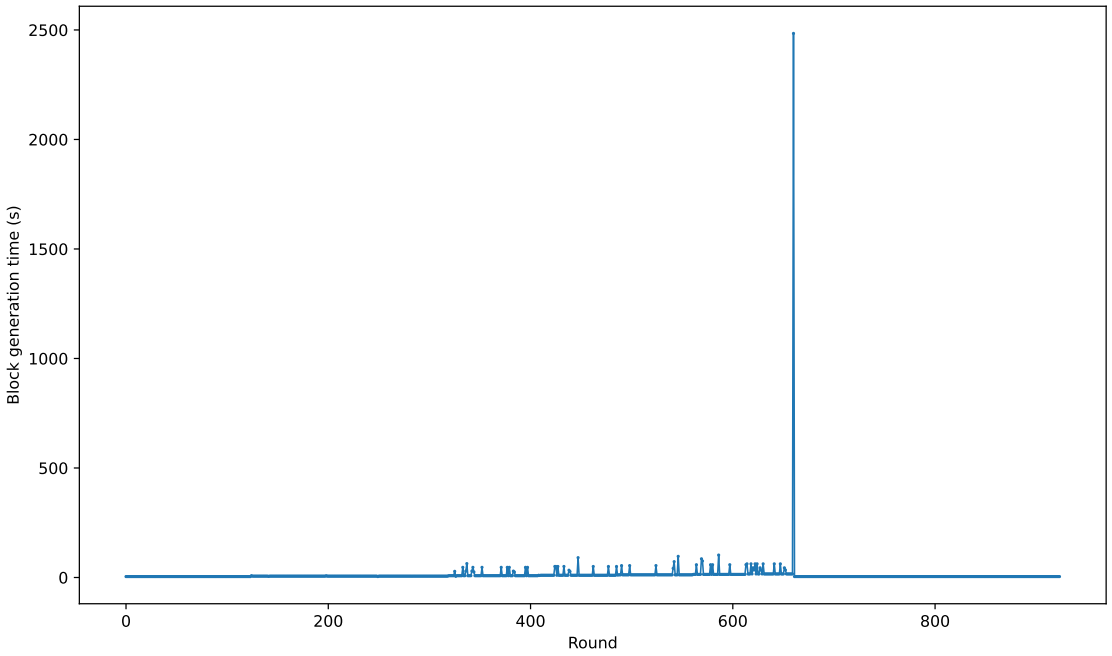


Fig. 3.35: Tiempo para la generación del bloque de cada ronda del experimento.

hacerlo para todos los enlaces del *relay*.

También resulta interesante el tiempo de recuperación de la red: en este caso es de 88 s, ampliamente superior que los casos anteriores. Sin embargo, tampoco resulta algo suficientemente grande como para afectar significativamente a la red, pero es notoria su mención.

Considerando esto, resulta ser la mejor opción para realizar un ataque, pero con la complejidad de que los nodos no se encuentran listados públicamente como los *relays*: solamente estos últimos conocen su IP al recibir la conexión del cliente de `Algorand` operando la cuenta. Si pudiéramos tener acceso a un *relay*, revisar las conexiones y tener la suerte de que los nodos que queremos atacar se encuentren conectados al mismo podríamos obtener sus direcciones IP e intentar un ataque directo a los nodos.

Como el *stake* del sistema real de `Algorand` se encuentra sumamente concentrado en esta etapa, no solo a nivel de las pocas cuentas que lo mantienen, sino que algunas de ellas delegan su *stake* a otras cuentas que terminan siendo operadas en un mismo nodo. Luego, hay nodos que manejan un *stake* muy alto de la red y serían críticos en el caso de recibir un ataque, perjudicando inmediatamente el avance de la red.

Sin embargo, algo que suponemos para estas redes es que los nodos solo están conectados a un *relay*, cuando en la realidad estos conocen el listado completo de *relays* de la red y eligen aleatoriamente cuatro de ellos para conectarse inicialmente. Luego, durante la ejecución del cliente, guardan métricas de cada conexión e intentan intercambiar al *relay* con menor desempeño por otro de los disponibles. Así, en el caso de que alguno de ellos esté fallando, es probable que el mismo nodo lo detecte, pruebe cambiando la conexión a otro *relay* y así evite quedar desconectado de la red.

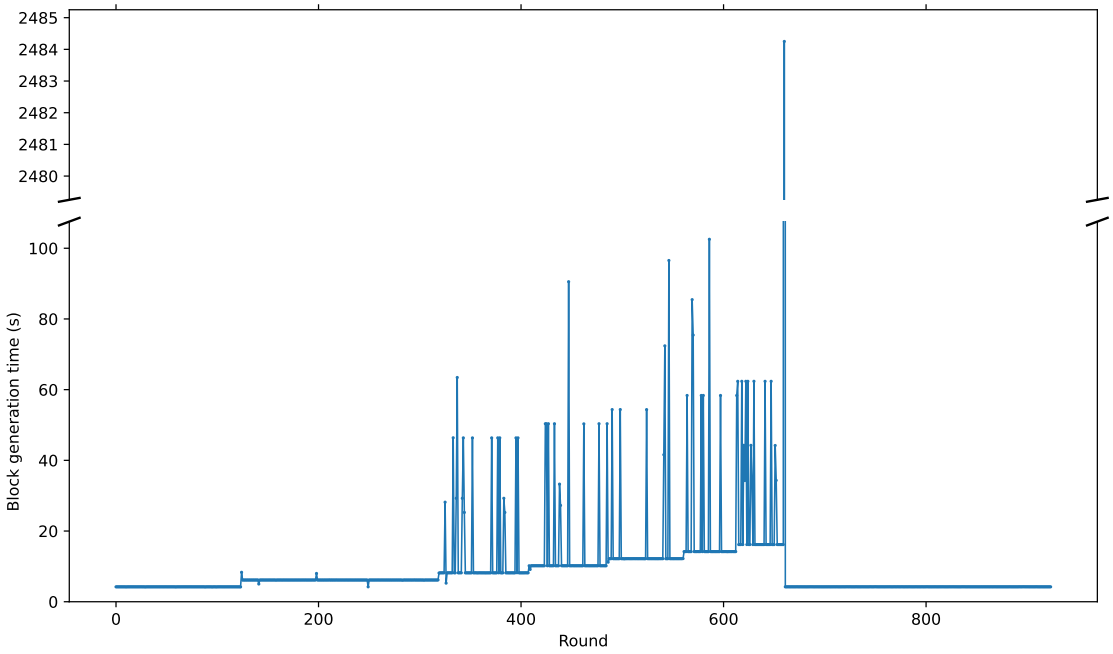


Fig. 3.36: (Detalle) Tiempo para la generación del bloque de cada ronda del experimento.

3.10. Red de *relays* y nodos en clique física: conexiones lógicas realistas

En la introducción mencionamos la arquitectura de la red lógica de `Algorand`, donde los nodos se conectan a cuatro *relays* del sistema y tratan de estar constantemente intercambiando al peor de ellos en cuanto a desempeño por otros, y así evitar perder conectividad. Con los experimentos anteriores forzamos que cada uno de los nodos esté conectado a un solo *relay* durante toda la duración del mismo, de manera que al aplicar los *delays*, el nodo no conoce otros para intentar conectarse, continuar en la red y así no afectar el progreso de la misma.

En base a esto, planteamos un experimento donde no limitamos el conocimiento de los nodos sobre la totalidad de los *relays* del sistema, y además agregamos una conexión física entre cada nodo y todos los *relays*, quedando una red como la que se observa en la imagen 3.37. Si bien, nuevamente, esta aproximación física no resulta realista, la idea será que si afectamos todos los enlaces de un *relay*, esto no impida que exista un camino sin demoras hacia los demás *relays*.

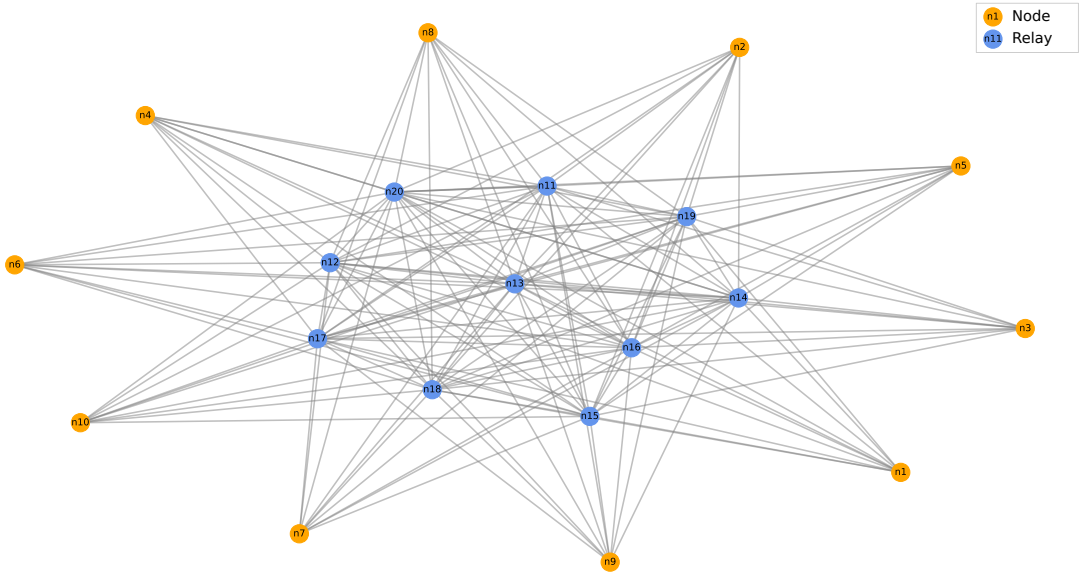


Fig. 3.37: Topología de la red física.

Para la red lógica, no predefinimos las conexiones entre todos los partícipes de la red, permitiendo que se conecten libremente entre ellos, queriendo averiguar cuál sería el comportamiento de la red real en el caso de que aplicáramos interrupciones en los *relays* del sistema.

Nos interesa conocer qué sucede si los *relays* del sistema van siendo atacados y demorados en su conexión, mientras que los nodos tienen la libertad de intercambiarlos por otros más saludables. En los experimentos anteriores la red se detenía debido a que al atacar estos *relays*, los nodos perdían la conexión a la misma y una parte necesaria del *stake* parecía desconectada. Aquí queremos ver si la red es lo suficientemente robusta como para tolerar el fallo de sus *relays* y qué cantidad de ellos pueden caer sin afectarla.

Afectamos incrementalmente todos los *relays* del sistema con un *delay* de 8 s, sabiendo en base a lo experimentado que para ese valor quedan inutilizables.

Sistema:

- **Nodos:** 10 (n1 a n10)
- ***Relays*:** 10 (n11 a n20)
- ***Relays* conectados por nodo:** 4 lógico, 10 físicos (20 ms)
- ***Relays* conectados por *relay*:** 4 lógicos, 9 físicos (30 ms)
- ***Stake*:** homogéneo
- **Tiempo de ejecución:** 190 min
- **Servidor:** local (1 servidor)
- ***Relays* afectados (según rango de minutos) con un *delay* de** 8 s**:**
  - 0 min a 10 min: ninguno
  - 10 min a 20 min: n20
  - 20 min a 30 min: n19 y n20
  - 30 min a 40 min: n18, n19 y n20
  - 40 min a 50 min: n17, n18, n19 y n20
  - 50 min a 60 min: n16, n17, n18, n19 y n20
  - 60 min a 70 min: n15, n16, n17, n18, n19 y n20
  - 70 min a 80 min: n14, n15, n16, n17, n18, n19 y n20
  - 80 min a 90 min: n13, n14, n15, n16, n17, n18, n19 y n20
  - 90 min a 100 min: n12, n13, n14, n15, n16, n17, n18, n19 y n20
  - 100 min a 110 min: n11, n12, n13, n14, n15, n16, n17, n18, n19 y n20
  - 110 min a 120 min: n12, n13, n14, n15, n16, n17, n18, n19 y n20

De esta manera, vamos dejando inutilizables uno por uno los *relays* del sistema, comenzando con el n20, hasta el n11, quedando ninguno disponible. En ese momento la red no debería poder progresar, pero luego activamos nuevamente el n11 de manera de verificar si con uno solo se recupera la misma.

Partiendo de los resultados observados en las figuras 3.38, 3.39 y 3.40, vemos que durante todo el experimento se mantiene constante la creación de bloques a pesar de todos los *relays* que son atacados, pudiendo verificar que los nodos al detectar tiempos de latencia altos en su conexión a ellos, deciden desecharlos y buscar uno que funcione correctamente. El único momento donde la red deja de funcionar es cuando ya no hay *relays* disponibles que no hayan sido atacados y los nodos quedan incomunicados, sin poder compartir sus mensajes de protocolo de consenso.

En el último tramo, cuando se vuelve a habilitar el *relay* n11, la red reanuda su curso y vuelve al mismo ritmo que tenía previamente. El tiempo de recuperación en este caso es de 27 s, valor superior a los tiempos que veíamos previamente, pero que resulta muy optimista para una red completamente incomunicada debido a estas altas latencias.

Con este experimento podemos ver que en un caso realista, atacar a un *relay* no sería algo eficaz para afectar la red: los nodos detectarían este problema y se conectarían a otro que esté funcionando correctamente. Por esto, resultaría mucho más efectivo poder atacar directamente a los nodos con mayor cantidad de participación en la red y replicar lo visto en el experimento de la sección anterior, donde afectamos la conexión entre el nodo y sus *relays*.

El problema en este caso entonces será poder encontrar a estos nodos para poder elegirlos como blanco de nuestro ataque, entendiendo que solo los *relays* a los que están conectados los conocen.

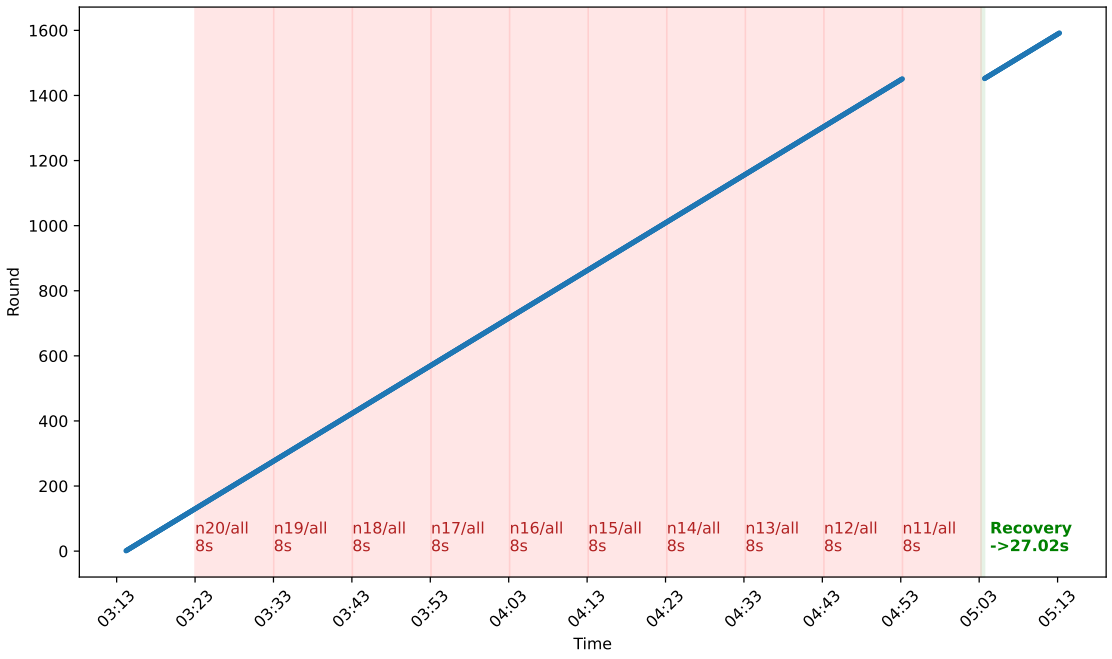


Fig. 3.38: Bloques producidos por la red en todo el experimento tomados del nodo n11, donde el área roja delimita el tiempo donde se aplica el *delay* entre los nodos indicados en la figura, mientras que el área verde se corresponde con el tiempo de recuperación de la red para retomar la producción de bloques.

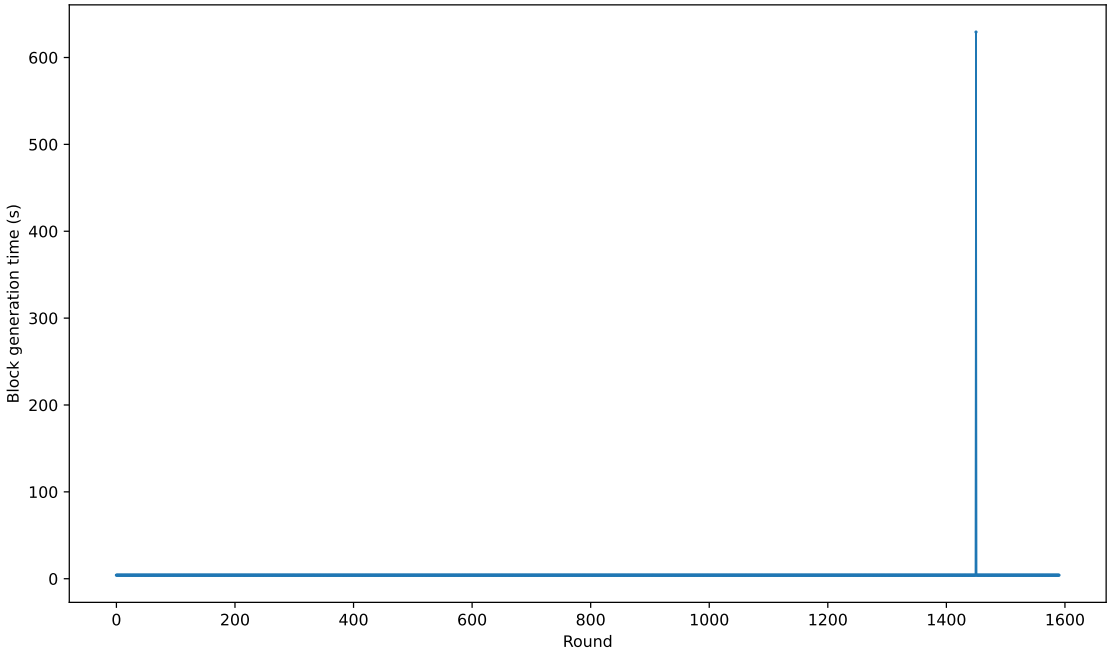


Fig. 3.39: Tiempo para la generación del bloque de cada ronda del experimento.

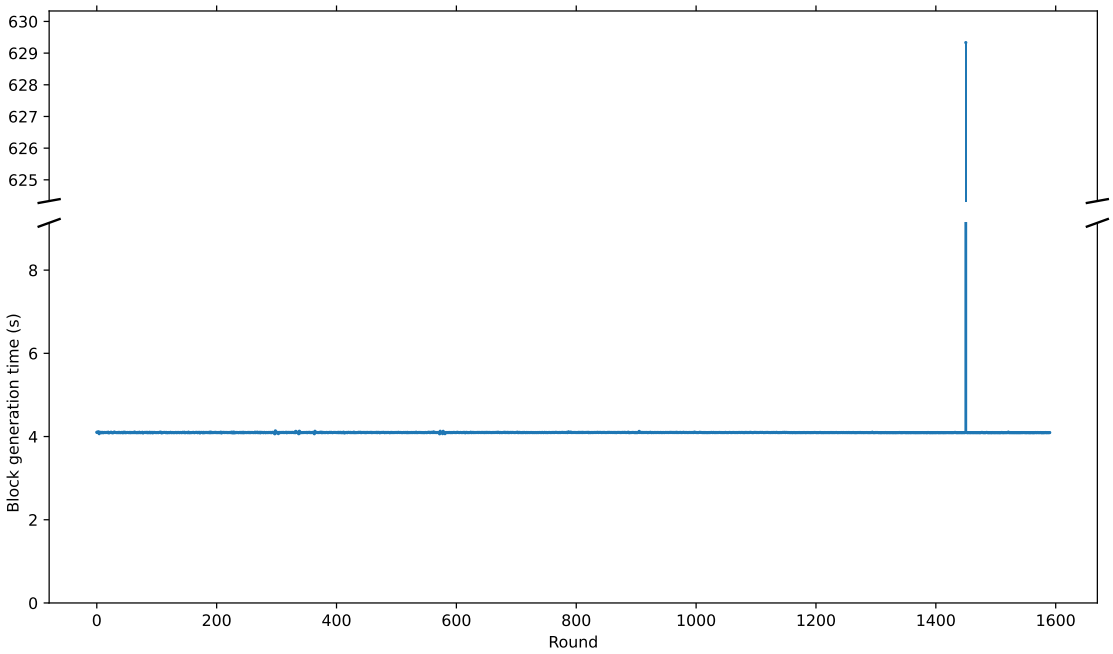


Fig. 3.40: (Detalle) Tiempo para la generación del bloque de cada ronda del experimento.

4. RECONSTRUYENDO LA TOPOLOGÍA LÓGICA

4.1. Introducción

Si bien la arquitectura general del sistema está definida, solo los *relays* son públicos en la misma. De manera que de antemano no podemos saber dónde se encuentra cada uno de los nodos participando del consenso de la red ni a qué *relays* se encuentran conectados en cada momento. En base a lo visto previamente, un atacante de la red, sabiendo dónde se encuentra cada nodo, podría potencialmente agregar demoras en los mismos y perjudicar el correcto avance del sistema. Luego, descubrir la topología en un momento dado (entendiendo que va mutando en el tiempo, pero que a fines prácticos la suponemos estática para nuestros experimentos) puede ser una puerta de entrada a potenciales ataques, sobre todo en etapas donde hay mucha concentración de *stake*: con los resultados anteriores, alcanzaría con afectar un 20 % del *stake* total para detener la red. Hoy en día hay cinco cuentas que controlan ese porcentaje de *stake*, con lo cual si pudiéramos encontrar qué nodos participan en nombre de ellas, podríamos potencialmente atacarlos y detener la red.

Sin embargo, sólo los *relays* a los que se conectan estos nodos pueden conocer cada una de las direcciones IP. Con lo cual nos planteamos un escenario donde se logra comprometer un único *relay* para lograr hacer escucha pasiva de los mensajes que recibe, es decir, no se agrega tráfico en la red para bajar las chances de detección. La idea será intentar reconstruir la topología de manera de poder identificar a qué *relays* están conectados los nodos que corren estas cuentas para así poder determinar los *relays* a atacar y encontrar la IP de los nodos que corren las cuentas más importantes. En el caso de que estos nodos estuvieran conectados directamente a nuestro *relay*, sería casi inmediato conocer la IP de los mismos. Por eso, nos resulta interesante el caso en el que eso no sucede. También cabe recordar que todos los *relays* del sistema son públicos y contamos con un listado de los mismos con su identificador e IP publicados.

Cabe destacar que la pobre descentralización del *stake* conlleva riesgos como éste: hay puntos muy críticos en la red que si fueran atacados (sin ser ataques complejos, como el visto en la parte anterior) podrían evitar el avance del sistema. Si la descentralización fuera más fuerte, incrementaría de gran manera el esfuerzo que requeriría lograr detenerla y esto sería desmotivante para el ataque.

Para esta parte nos planteamos dos escenarios a analizar: por un lado, una red completamente descentralizada, donde el *stake* se reparte equitativamente entre todos los participantes, y otro escenario donde la distribución del *stake* respeta la actualidad de la red (aprovechando el hecho que es público el *stake* de cada cuenta y si participa o no del consenso, activamente o delegando su *stake* a otra cuenta).

Al igual que en la primer parte, asignamos una cuenta por nodo para facilitar el análisis de datos, entendiendo que la cantidad de mensajes en la red que emite un nodo depende del *stake* de las cuentas que maneja. Luego, el nodo no es más que una instancia del cliente ejecutándose para ciertas cuentas y podemos simplificarlo al pensar que en realidad es la misma cuenta la que envía los mensajes. Así, cuando nos referimos a un nodo, también estamos haciendo referencia a la cuenta que está corriendo.

4.2. Mensajes de la red

Como primer paso para conocer la factibilidad de encontrar a cada nodo dentro de la red atacando silenciosamente un *relay*, debemos analizar los mensajes que se envían en la misma y qué información trae cada uno. Buscamos algún mensaje que nos dé información con respecto a la IP del nodo corriendo nuestras cuentas objetivo.

Dado que no es algo documentado para la red, analizamos a nivel de código la implementación del sistema de comunicación entre los nodos. Allí encontramos la clase *IncomingMessage* que encapsula de manera genérica todos los mensajes que intercambia el sistema. La misma mantiene la siguiente estructura principal (se ocultan campos adicionales que no nos interesan):

```
type IncomingMessage struct {
    Sender Peer
    Tag    Tag
    Data   []byte
    Received int64
    ...
}
```

El campo *Sender* hace referencia, evidentemente, al nodo que envió el mensaje. Aquí encontramos toda la información del mismo, como la IP y otros identificadores. El campo *Tag* es un enumerado que facilita la identificación del tipo de mensaje, siendo *Data* el contenido del mensaje cuya estructura dependerá del tipo de mensaje. Así, la interfaz de los mensajes es siempre la misma, siendo el *handler* de cada tipo el que conoce realmente la estructura del contenido. Finalmente el campo de *Received* es un *unix timestamp* del momento en que fue recibido el mensaje.

Con esta implementación se facilita muchísimo la intercepción de los mensajes para conocer su distribución, dado que podemos agregar código en el *handler* que los recibe antes de ser procesados en base a su *tag* y *loggear* el tipo de cada mensaje recibido. Lo negativo es que el contenido del mensaje lo conoce internamente el *handler* de cada tipo de mensaje, que en varios casos se encuentran en paquetes diferentes y dificulta hacerlo en un lugar unificado (porque básicamente estamos rompiendo el patrón de diseño establecido).

En cuanto al campo *Tag*, cuenta con 15 valores posibles, algunos ya deprecados o con poco uso, entre ellos los más destacados son:

- "AV" (*AgreementVoteTag*): es enviado por los partícipes del consenso para ejercer un voto. En cada ronda, cada uno de los nodos participantes va a enviar un conjunto de ellos dependiendo de su *stake* (recordemos que las cuentas con mayor *stake* emiten mayor cantidad de votos). Entre la información más importante del mensaje encontramos información de la ronda sobre la que se está votando y quién es la cuenta emitiendo el voto.

- "NP" (*NetPrioResponseTag*): es utilizado para manejar prioridades de conexión en base a un protocolo de *challenge-response*. Contienen mucha información a nivel del red y *metadata* sobre el nodo con el que se establece la conexión, pero dada su función solo los recibimos de nodos con los que nos conectamos.

- "PP" (*ProposalPayloadTag*): contiene un bloque propuesto para la ronda, con todo lo necesario para ser verificado por otros participantes. Contiene el identificador de la cuenta haciendo la propuesta y pruebas criptográficas de la misma.

- "TX" (*TxnTag*): está compuesto por un listado de transacciones firmadas que fueron recibidas por el nodo que generó el mensaje. Este tipo cuenta con varios subtipos, por ejemplo para transacciones de pagos, de registración de claves para la participación en el consenso, transferencia de *assets*, etc. A su vez contiene la información criptográfica necesaria para verificar estas transacciones y el identificador de la cuenta que las firmó. También encontramos el *tag* "*tx*" (en minúsculas) que representa una nueva implementación de este tipo de mensajes, de manera que lo trataremos de la misma manera.

- "UE" (*UniEnsBlockReqTag*): la utilizan los nodos para hacer un *catch-up* con la red (en caso de desincronizarse o al inicio de la misma). Su estructura es muy básica y se lo utiliza solamente con el fin de recuperarse en caso de perder el ritmo de la red.

- "VB" (*VoteBundleTag*): es un *bundle* que contiene los votos emitidos y la *metadata* de la siguiente ronda. Básicamente es un resumen del estado de la votación sobre el consenso para un momento dado de la ronda.

Para la elección del tipo de mensaje que queremos analizar tenemos dos requisitos importantes: necesitamos que contenga el identificador de la cuenta que lo generó y también que aparezcan múltiples mensajes a lo largo del desarrollo del sistema (es decir, que sea un mensaje periódico y predecible). El primer requerimiento nos va a permitir asociar un mensaje a una cuenta, y así intentar reconstruir dónde se encuentra conectado este nodo. El segundo nos permite hacer análisis más precisos en cuanto a la cantidad de información necesaria en base a rondas como para poder reconstruir la red: no queremos depender de que la red requiera transacciones para poder hacerlo, queremos poder lograrlo con mensajes periódicos de la misma.

El primer requerimiento lo cumplen los mensajes de tipo *AV*, *PP* y *TX*, en ellos hay un mensaje firmado por una cuenta y se adjunta su identificador. En el caso de *NP*, podemos conocer la IP del nodo al que nos estamos conectando, pero no tenemos información de las cuentas que está corriendo el mismo. Para el *UE*, no se encuentra tampoco información de las cuentas corriendo en el nodo, es solo un mensaje de resincronización. Finalmente, en el *VB* tenemos un resumen de todos los votos, pero al ser algo general perdemos la trazabilidad de dónde vino cada uno de ellos.

En cuanto al requerimiento de periodicidad, vemos que los mensajes de tipo *AV*, *PP* y *VB* lo cumplen: son parte del consenso de cada ronda, de manera que si no estuvieran, no sería capaz de avanzar. Nuevamente, *NP* no lo cumple dado que solo aparecen cuando un nodo realiza una nueva conexión (lo cual sucede al comienzo de la red y luego puede suceder esporádicamente por múltiples motivos). Tampoco lo cumple *TX*, estos mensajes si bien aparecen a pesar de que no se produzcan transacciones en el sistema, escalan mucho si existen. Y *UE* no lo cumple por lo visto previamente, solo los veremos cuando se inicia la red o cuando hay algún problema de conectividad en alguno de los nodos.

Cabe aclarar que estos mensajes los vemos desde los *relays*, los nodos normales se pierden gran parte de los mensajes de la red ya que solo envían sus propios mensajes a los *relays* y luego reciben una especie de *digest* de parte de los *relays* a los que están conectados

para actualizar su estado. Es por esto que no nos resulta de interés su análisis: no podemos ver los mensajes que envían los demás *relays*, recibimos información ya procesada.

Para validar lo descrito previamente, realizamos un experimento corriendo 30 nodos y 10 *relays* durante 900 rondas (aprox. una hora) y analizamos los distintos tipos de mensajes recibidos en uno de los *relays* del sistema. Planteamos las topologías física y lógica de esquematizadas en las figuras 4.1 y 4.2 respectivamente.

Sistema:

- **Nodos:** 30 (n1 a n30)
- ***Relays*:** 10 (n31 a n40)
- ***Relays* conectados por nodo:** 4 lógicos, 4 físicos
- ***Relays* conectados por *relay*:** 4 lógicos, 4 físicos
- ***Stake*:** homogéneo
- **Tiempo de ejecución:** 900 rondas (1 h)
- **Servidor:** local (1 servidor)

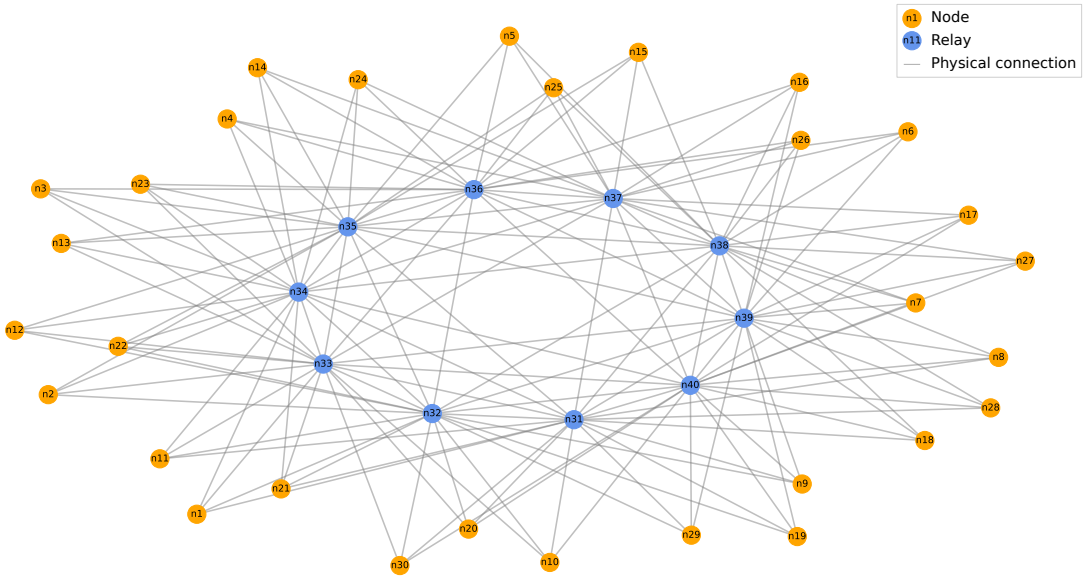


Fig. 4.1: Topología de la red física (enlaces en gris).

Con este sistema obtuvimos la distribución de mensajes planteada para uno de los *relays* del sistema (para el resto, la distribución es similar).

En la figura 4.3 graficamos la cantidad de mensajes recibidos por el *relay* por tipo de *tag*. Encontramos una gran cantidad de mensajes de tipo *tx* (por más que no estamos inyectando transacciones de *tokens*, pero este conjunto no solo abarca ese tipo de transacciones según lo ya visto), otra cantidad importante de mensajes AV y luego los demás tienen una aparición muy baja (siendo completamente despreciable frente a los dos primeros).

No solo nos interesa la cantidad de mensajes de cada tipo en la ejecución del sistema, sino que queremos un tipo de mensaje que se mantenga relativamente constante en ella. Analizamos entonces la cantidad de mensajes en la línea temporal de la corrida:

En 4.4 observamos que la cantidad de mensajes de tipo *tx*, AV y PP se mantienen relativamente constantes en el tiempo (no hay saltos significativos en ninguno de ellos).

También es interesante observar la aparición de los mensajes de tipo *UE* al principio del sistema, cuando los nodos están comenzando a conectarse entre ellos y emiten ese mensaje para sincronizarse con los demás. Y por otro lado, los mensajes de tipo *NP* cuando el *relay* analizado realiza una nueva conexión a otro nodo o *relay*.

En base a estos resultados, decidimos optar por los mensajes de tipo *AV*, cumplen con todo lo que buscábamos: mensajes que nos permiten identificar a la cuenta que los

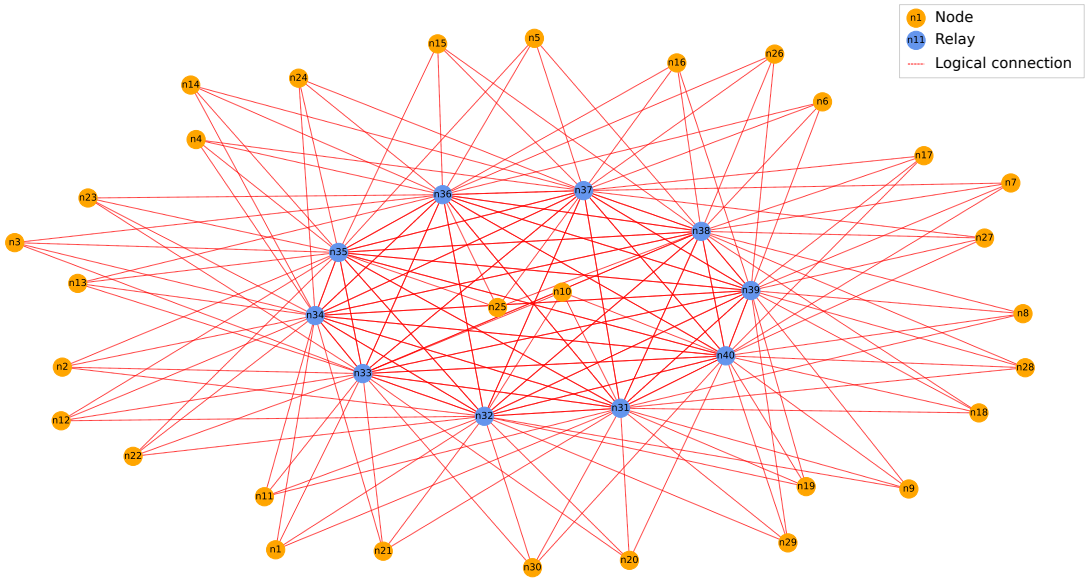


Fig. 4.2: Topología de la red lógica (enlaces en rojo).

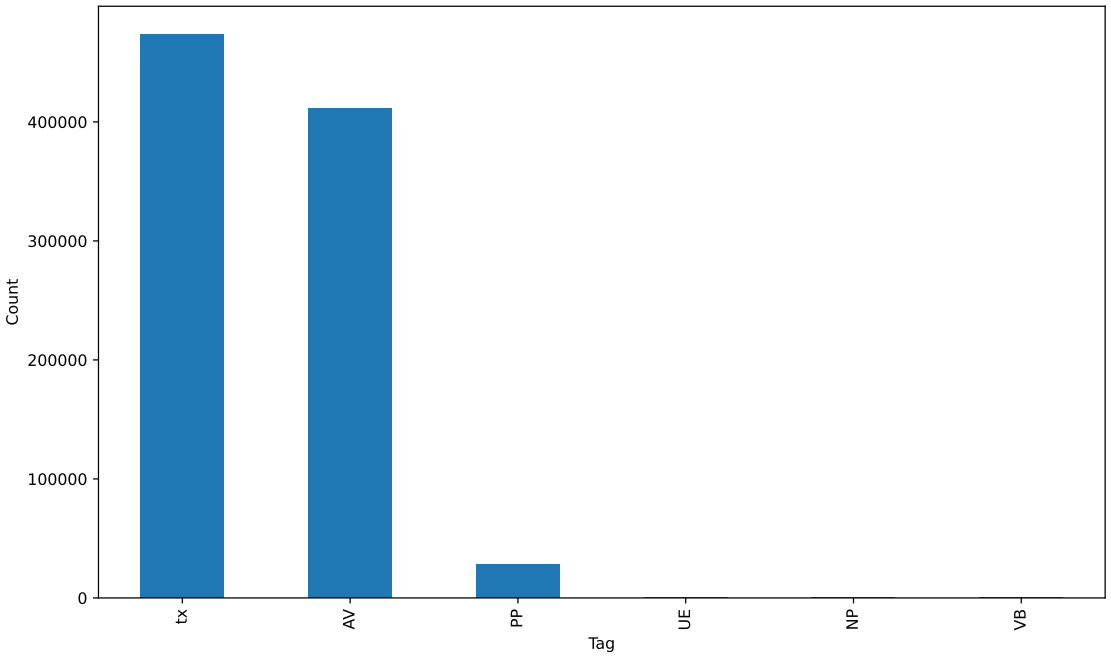


Fig. 4.3: Cantidad total de mensajes recibidos, para los tipos AV, NP, PP, UE, VB y tx.

firma para poder deducir la ubicación del mismo y que haya una cantidad consistente en el tiempo que nos dé esa información.

Por otro lado, también graficamos los tipos de mensaje recibidos de cada uno de los nodos que se conectaron al *relay* en la figura 4.5.

Todos los nodos con numeración menor a 31 son nodos participantes del consenso que están conectados directamente al *relay* analizado, mientras que los superiores son otros *relays* desde los que recibimos información. Es interesante notar que hay algunos de estos *relays* con los que se intercambia más información que con otros, dado que el *relay* establece conexiones activas con algunos de ellos, mientras que de otros los recibe de manera más pasiva. También podría cambiar conexiones por otros *relays*, aumentando la información recibida de cada uno de ellos. Por este motivo, podría suceder que en ciertas ventanas de tiempo un *relay* esté conectado a algunos *relays* y luego cambie en otra, de manera que nos resulta muy importante tomar distintas ventanas para mejorar los resultados de nuestro análisis.

Por una cuestión de desempeño, nos interesa hacer nuestros experimentos sin necesidad de inyectar transacciones en el sistema: como vamos a trabajar con mensajes de tipo AV, no nos interesan los de tipo *tx*. Y como la inyección de las transferencias involucra tener procesos corriendo constantemente para que se envíen a los nodos correspondientes a tiempo, preferimos evitar ese consumo de recursos. Para eso, queremos ver que el mismo con o sin transacciones mantiene la misma cantidad de mensajes de tipo AV en el tiempo (es decir, las transacciones no afectan a la cantidad de mensajes AV). Realizamos otro experimento donde inyectamos transacciones en el sistema de manera incremental. Primero 60 transacciones por ronda a lo largo de 100 rondas, luego esperamos 50 rondas para que se estabilice el sistema. Después inyectamos 300 transacciones por ronda durante 100 rondas,

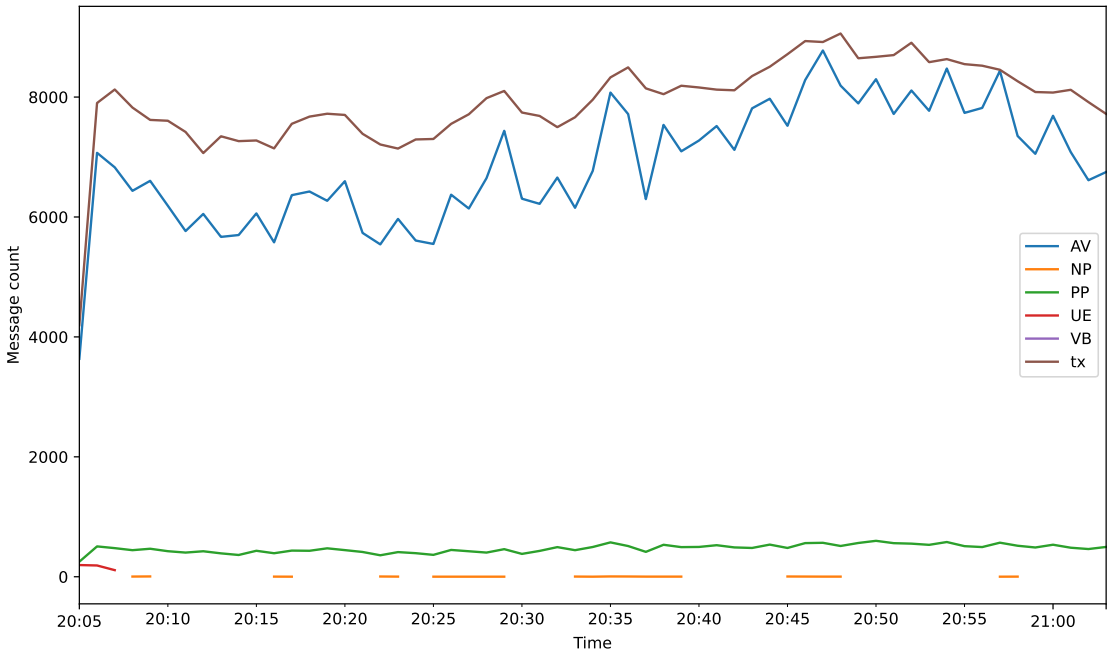


Fig. 4.4: Cantidad de mensajes recibidos durante la duración del experimento, para los tipos AV, NP, PP, UE, VB y tx.

esperamos 50 rondas. Finalmente inyectamos 1500 por ronda nuevamente por 100 rondas y volvemos a repetir lo mismo decrementando la cantidad de transacciones a los valores anteriores.

Sistema:

- **Nodos:** 30
- ***Relays*:** 10
- ***Relays* conectados por nodo:** 4 lógicos, 4 físicos
- ***Relays* conectados por *relay*:** 4 lógicos, 4 físicos
- ***Stake*:** homogéneo
- **Tiempo de ejecución:** 750 rondas (50 min)
- **Transacciones por ronda (tpr):**
  - Ronda 50 a 150: 60 tpr.
  - Ronda 200 a 300: 300 tpr.
  - Ronda 350 a 450: 1500 tpr.
  - Ronda 500 a 600: 300 tpr.
  - Ronda 650 a 750: 60 tpr.
  - Las rondas fuera de los rangos presentados se corren sin inyección de transacciones para asegurarnos de que todas las pendientes sean procesadas antes del siguiente rango.
- **Servidor:** local (1 servidor)

Con esta corrida, vemos en la figura 4.6 que no hay diferencia en la cantidad de mensajes *tx* para el caso de 60 o 300 *txs* por ronda (dado que dentro del mismo mensaje los nodos pueden enviar todas las transacciones recibidas en esa ronda), pero se produce un pico

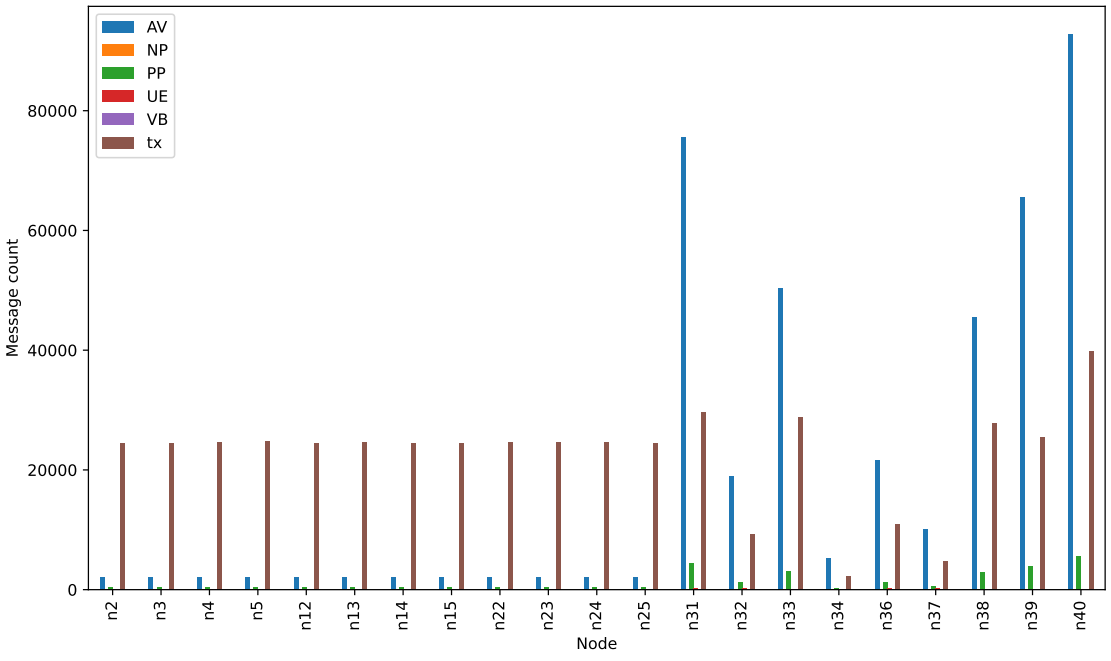


Fig. 4.5: Cantidad total de mensajes recibidos de cada nodo, para los tipos AV, NP, PP, UE, VB y tx.

importante cuando se inyectan 1500 por ronda. Vemos que en la ejecución del sistema la cantidad de mensajes *AV* se mantiene relativamente constante, siendo que no es afectada cuando la cantidad de mensajes *tx* crece al doble.

Luego, podemos avanzar con la experimentación sin necesidad de inyectar transacciones en el sistema sabiendo que la presencia de ellas no modifica el análisis de los mensajes *AV* y nos permite ahorrar los recursos que serían necesarios para su inyección.

4.3. Redes experimentales

En estos experimentos la idea será correr una instancia de la red con un cuatro tamaños distintos, dejar que progrese durante una cantidad de bloques y luego hacer análisis de los mensajes enviados en la misma. Después de la corrida, elegimos un *relay* como candidato a ser el atacado y analizamos los mensajes que recibió del resto de los *relays*. Con ellos intentamos reconstruir la topología y validamos si la topología encontrada coincide con la real y en qué medida.

Se plantearon redes donde los nodos se encuentran equidistribuidos entre los *relays*: cada uno de ellos se conecta a cuatro *relays* predefinidos y los mantienen durante toda la ejecución del experimento, de manera de tener control de la estructura lógica de la red (sino, podrían estar cambiando de *relays* durante la misma y complicaría luego el análisis de la red). Los *relays* también se configuran de la misma manera, predefiniendo cuáles van a ser los otros cuatro *relays* con los que va a intercambiar mensajes durante la duración del experimento, en base a lo visto previamente con respecto a las conexiones que realiza un *relay* al no ser limitado en este aspecto.

Todos los experimentos se repiten tres veces para validar lo obtenido en la primera

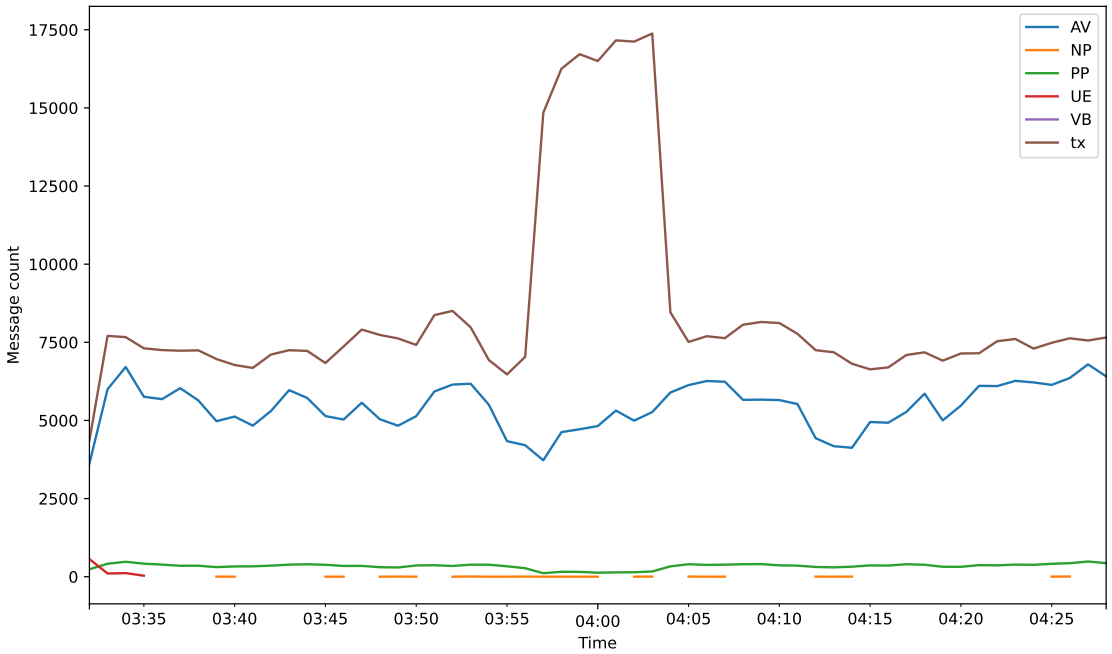


Fig. 4.6: Cantidad de mensajes recibidos durante la duración del experimento con transacciones, para los tipos AV, NP, PP, UE, VB y tx.

ejecución, obteniendo variaciones mínimas en los resultados en base a los factores aleatorios de la VRF, pero que no alteran los resultados finales. Por este motivo, solo se reportan los resultados de una de estas repeticiones.

4.3.1. Tamaños de corrida

Como fue descrito en la introducción, la red cuenta actualmente con alrededor de 200 cuentas participando activamente del consenso y 106 *relays*. Al principio de esta investigación, la cantidad de cuentas era más cercana a las 300, con lo cual decidimos trabajar con una relación de tres a uno: por cada *relay* creamos tres nodos/cuentas. De esta forma, analizamos cuatro tamaños de red:

- Pequeña (30 nodos y 10 *relays*): nos permite realizar análisis con una cantidad controlable de información al ser un sistema chico para evaluar cómo se comporta el sistema y poder sacar conclusiones al respecto.
- Mediana (60 nodos y 20 *relays*): duplicamos el tamaño anterior para analizar si se mantienen los comportamientos observados con mayor cantidad de nodos y *relays*.
- Semi-realista (150 nodos y 50 *relays*): escalamos la red a la mitad de la red real, para tener un primer acercamiento a la escala real, teniendo un caso intermedio entre lo analizado previamente y la realista.
- Realista (300 nodos y 100 *relays*): representa la actualidad de la red y nos permite ver si lo propuesto funcionaría en ella.

Con estos cuatro tamaños intentamos mantener una línea de trabajo que nos permita primero aprender con dos casos pequeños, luego validar lo aprendido en un sistema mayor y finalmente verificar si funcionaría en una red con escala real.

Como fue aclarado en la metodología (y luego será aclarado en cada detalle de sistema), los experimentos pequeños y medianos se corren localmente en equipos propios, mientras que los semi-realista y realista se corren en CloudLab debido a los requisitos de hardware que implica correr esa cantidad de clientes. En caso contrario, se aclara en la sección "Sistema" el servidor utilizado en cada uno.

4.4. Análisis inicial

4.4.1. Cantidad de mensajes por cuenta: conectadas vs. no conectadas

Nuestro primer acercamiento a este experimento plantea un escenario homogéneo donde todas las cuentas participan del consenso con el mismo peso. Con esta distribución, la red se encuentra completamente descentralizada y cada uno de las cuentas tiene la misma probabilidad de tomar decisiones en la misma. Si bien no representa el escenario más realista, lo tomamos como punto de partida para iniciar el análisis y explorar potenciales soluciones al problema planteado.

Comenzamos con una red pequeña, con 30 nodos y 10 *relays*, donde cada nodo se conecta a cuatro *relays* de la red (es decir, cada *relay* se conecta a 12 nodos). Estas conexiones están configuradas en cada uno de ellos de manera de que no se conecten a otros *relays* y podamos mantener el control de las conexiones de la red. Así, podemos analizar posteriormente los datos obtenidos ya que sabemos de antemano cuáles eran los

relays a los que se conectó el nodo (algo que se complicaría si dejáramos libre al nodo). También, en base a lo analizado en la documentación, código y la comunidad de `Algorand`, los nodos no deberían cambiar de *relays* constantemente, solo lo harían en caso de que alguno de ellos se degrade o baje su desempeño.

Sistema:

- **Nodos:** 30 (n1 a n30)
- ***Relays*:** 10 (n31 a n40)
- ***Relays* conectados por nodo:** 4 lógicos, 4 físicos
- ***Relays* conectados por *relay*:** 4 lógicos, 4 físicos
- ***Stake*:** homogéneo
- **Tiempo de ejecución:** 6 h
- **Servidor:** local (1 servidor)

En esta corrida, parándonos en el *relay* n35 (elegido aleatoriamente), recibimos del *relay* n31 la siguiente distribución de mensajes. Las cuentas resaltadas son aquellas que se encuentran efectivamente conectadas al *relay* n31. La cantidad de mensajes mostrados se corresponde con la corrida completa.

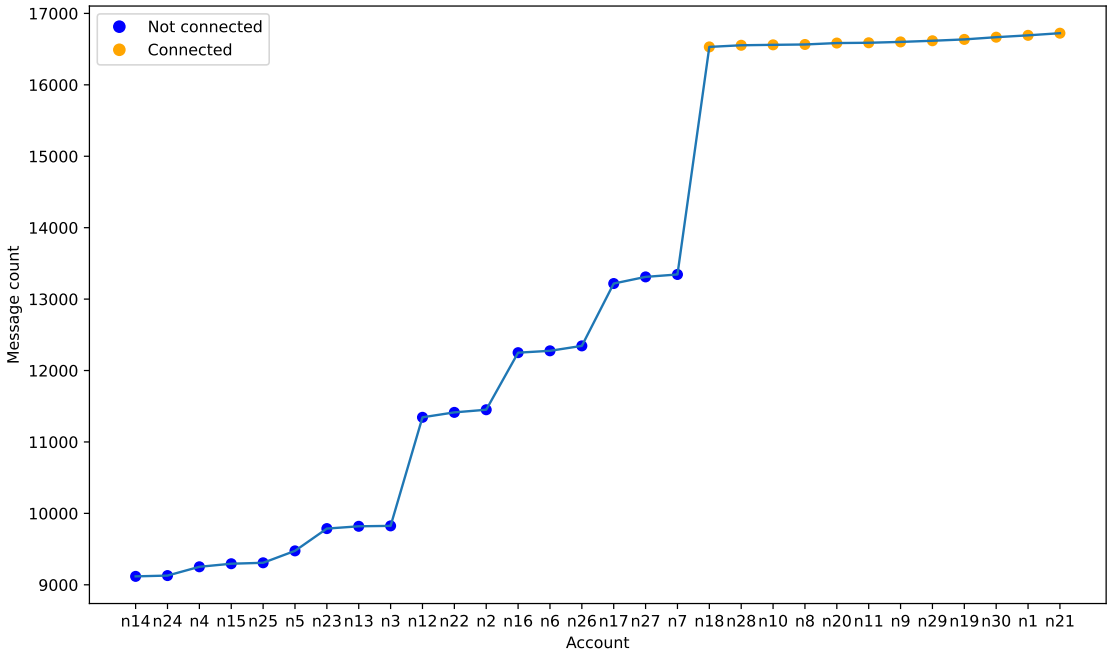


Fig. 4.7: Mensajes recibidos por nodo del *relay* n31 en todo el experimento, marcados con color azul aquellos no conectados y con naranja los conectados.

A partir de la tabla 4.1 y la figura 4.7, podemos ver que recibimos mayor cantidad de mensajes de las cuentas que están efectivamente conectadas al *relay* analizado (del n21 al n18 en la tabla): la cantidad de mensajes es similar entre ellos y luego notamos un decremento para los que no están conectados. El salto entre la cuenta n18 y la n27 implica una disminución de más de un 20 % en la cantidad de mensajes, algo considerable y que marca un quiebre en esos valores. Sin embargo, nos interesa también conocer estas diferencias en intervalos más pequeños de tiempo, queriendo ver si este quiebre también se presenta o es algo que requiere analizar toda la corrida.

sender *relay*	account	message count
n31	**n21**	16723
n31	**n1**	16693
n31	**n30**	16667
n31	**n19**	16636
n31	**n29**	16617
n31	**n9**	16601
n31	**n11**	16589
n31	**n20**	16585
n31	**n8**	16565
n31	**n10**	16560
n31	**n28**	16554
n31	**n18**	16531
n31	n7	13345
n31	n27	13311
n31	n17	13218
n31	n26	12345
n31	n6	12276
n31	n16	12250
n31	n2	11451
n31	n22	11414
n31	n12	11345
n31	n3	9825
n31	n13	9819
n31	n23	9787
n31	n5	9475
n31	n25	9308
n31	n15	9295
n31	n4	9252
n31	n24	9129
n31	n14	9118

Tab. 4.1: Cantidad de mensajes por cuenta y *relay* recibidos en el *relay* n35, para el sistema de 30 nodos y 10 *relays* con *stake* homogéneo.

La misma diferencia la vemos en los mensajes recibidos desde los demás *relays*, en todos con una diferenciación clara entre conectados y no conectados, pero no responden siempre al mismo porcentaje, sino que es variable para cada caso. Podemos detectar la diferencia analizando cada caso, pero no podemos determinar a priori un criterio para establecer ese corte.

Tomamos ventanas de diez minutos del experimento y analizamos los mensajes recibidos de cada uno de los *relays* a los que nos conectamos en el caso de uno de los *relays* del sistema.

En las figuras 4.8 y 4.9 podemos observar la cantidad de mensajes recibidos por cada uno de los nodos del sistema, desde dos *relays* (el n31 en la primera, el n32 en la segunda). Las columnas en naranja se corresponden con nodos que están efectivamente conectados al *relay* que envía la información, mientras que las azules son aquellos que no.

Es notable la diferencia que observamos en cantidad entre aquellos nodos que están conectados directamente al *relay* que envía los mensajes versus los que no lo están, lo

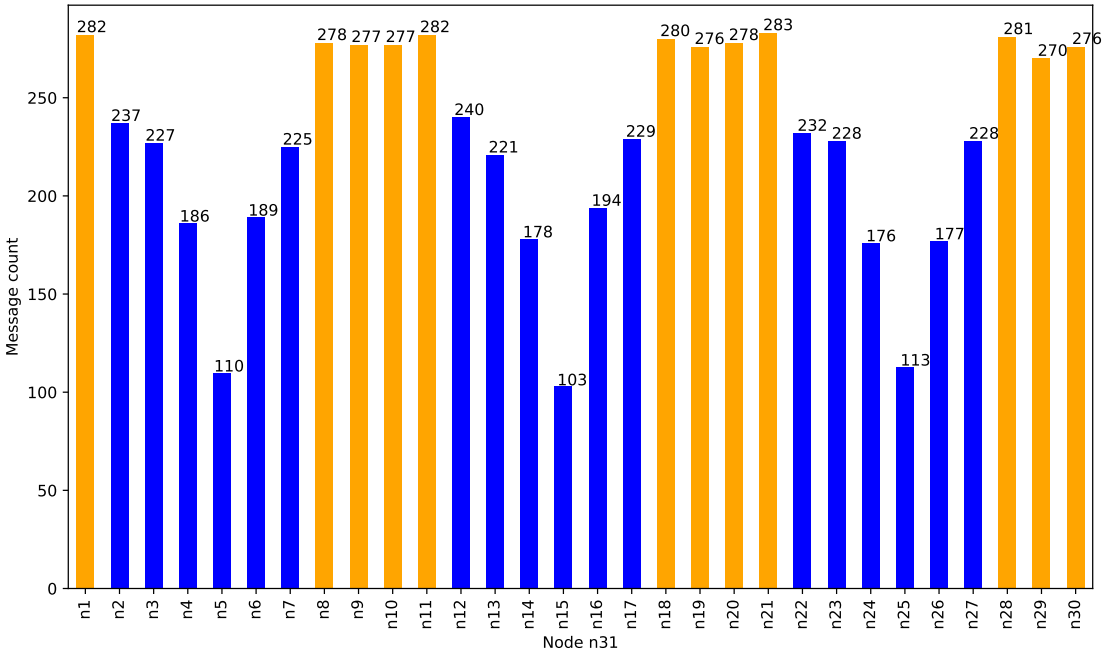


Fig. 4.8: Mensajes recibidos por nodo del *relay* n31 en una ventana de 10 minutos, marcados con color azul aquellos no conectados y con naranja los conectados.

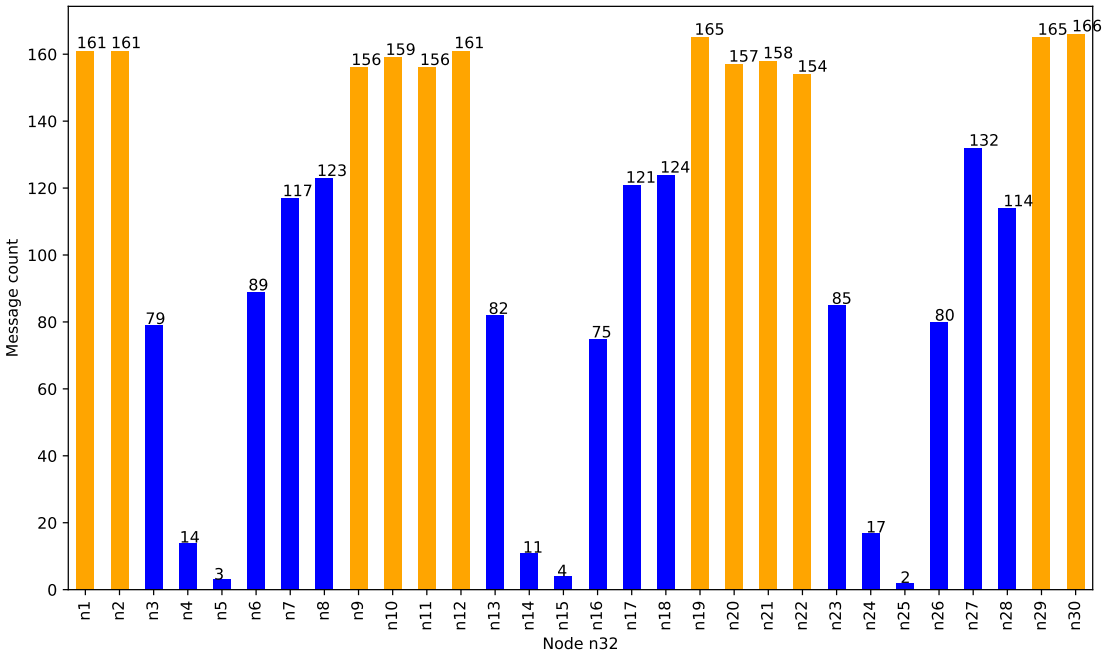


Fig. 4.9: Mensajes recibidos por nodo del *relay* n32 en una ventana de 10 minutos, marcados con color azul aquellos no conectados y con naranja los conectados.

cual nos da la pauta de que en el sistema, los *relays* nos van a dar más información de los primeros nodos. Con esto, para poder reconstruir la estructura de la red, podríamos estudiar una manera de distinguir la diferencia entre la cantidad de mensajes recibidos de parte de un *relay* entre los diferentes nodos y así encontrar un criterio que nos permita discriminar los conectados de los no conectados.

Esta tendencia se mantiene en el tiempo de la ejecución del sistema, siendo muy clara en algunas ventanas, aunque en otras no lo es tanto.

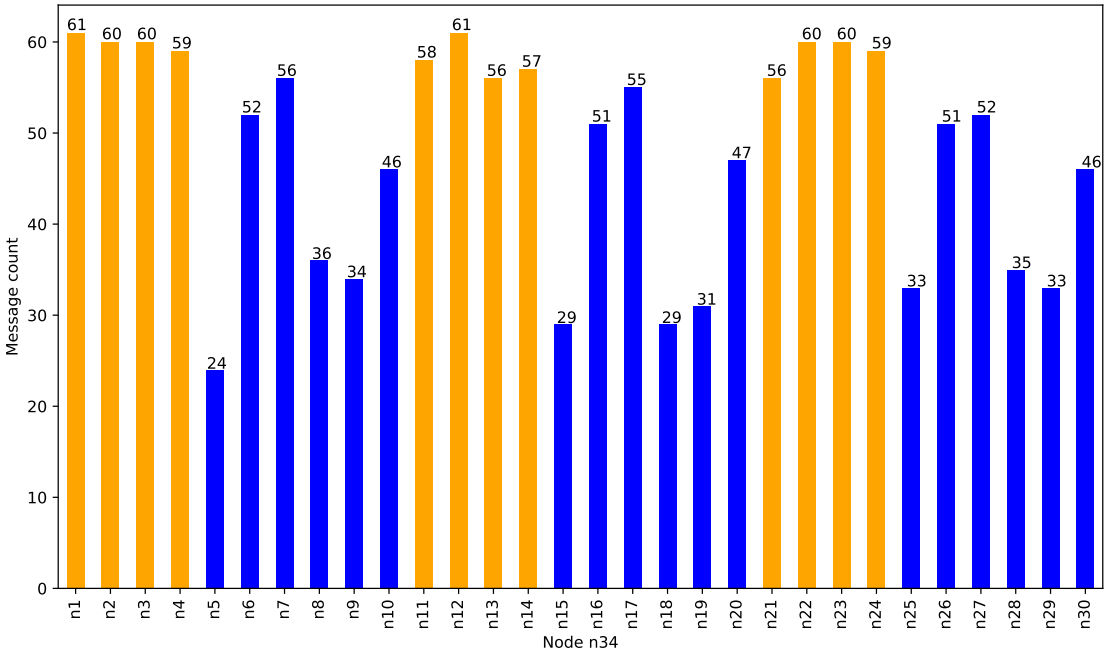


Fig. 4.10: Mensajes recibidos por nodo del *relay* n34 en una ventana de diez minutos, marcados con color azul aquellos no conectados y con naranja los conectados.

Con el ejemplo de la figura 4.10, puede verse que la distinción entre los nodos no es tan clara, con lo cual podría afectar nuestros análisis. Por esto, debemos ser cuidadosos al analizar las ventanas de información y no quedarnos con cada una de ellas como una verdad absoluta, sino hacer un trabajo de procesamiento de las ventanas y sacando conclusiones con un conjunto de ellas.

En primera instancia, esto nos da la pauta de que predecir los nodos conectados al *relay* es algo posible, pero siempre que analicemos una cantidad adecuada de ventanas para poder hacer esta predicción.

En otros casos, como el de la figura 4.11, también llega a suceder que los nodos no conectados tienen cantidades superiores a algunos de los conectados, en otros casos hasta superándolos, de manera que es otro punto a evaluar al intentar predecir las conexiones.

4.4.2. Disminuyendo cantidad de *relays* por nodo

Otro factor a considerar es la cantidad de *relays* a los que se encuentra conectado cada nodo durante su ejecución: si bien en la red real esta cantidad es como máximo cuatro (podrían ser menos, aunque en ese caso el nodo estaría intentando conectarse a nuevos nodos hasta llegar a los cuatro), nosotros podríamos definir el límite que queramos. En

la experiencia anterior con 30 nodos y 10 *relays*, conectamos cada nodo a cuatro *relays*, de manera que al estar distribuidos equitativamente, cada *relay* recibía la conexión de 12 nodos.

Queremos verificar que lo observado previamente se mantiene en el caso de que el nodo haga menos conexiones, por lo que realizamos pruebas donde los nodos se conectan a tres, dos y un *relays* cada uno. Nuestra intención es que si obtenemos el mismo resultado anterior para menor cantidad de conexiones, podemos suponer que en un escenario real donde los nodos podrían estar conectados a menos *relays* seguiría cubierto por nuestra experimentación.

Primero experimentamos conectando cada nodo a tres *relays* de la red, obteniendo lo siguiente:

Sistema:

- **Nodos:** 30 (n1 a n30)
- ***Relays*:** 10 (n31 a n40)
- ***Relays* conectados por nodo:** 3 lógicos, 3 físicos
- ***Relays* conectados por *relay*:** 4 lógicos, 4 físicos
- ***Stake*:** homogéneo
- **Tiempo de ejecución:** 6 h
- **Servidor:** local (1 servidor)

Vemos que en el caso de la tabla 4.2 y su figura asociada 4.12, el *relay* elegido (n40) recibe la conexión de nueve nodos, confirmando nuestra configuración de tres *relays* por nodo. Para estos nodos notamos nuevamente una cantidad de mensajes recibidos similar y superior al resto de los nodos.

Repetimos la experiencia, pero con dos *relays* por nodo:

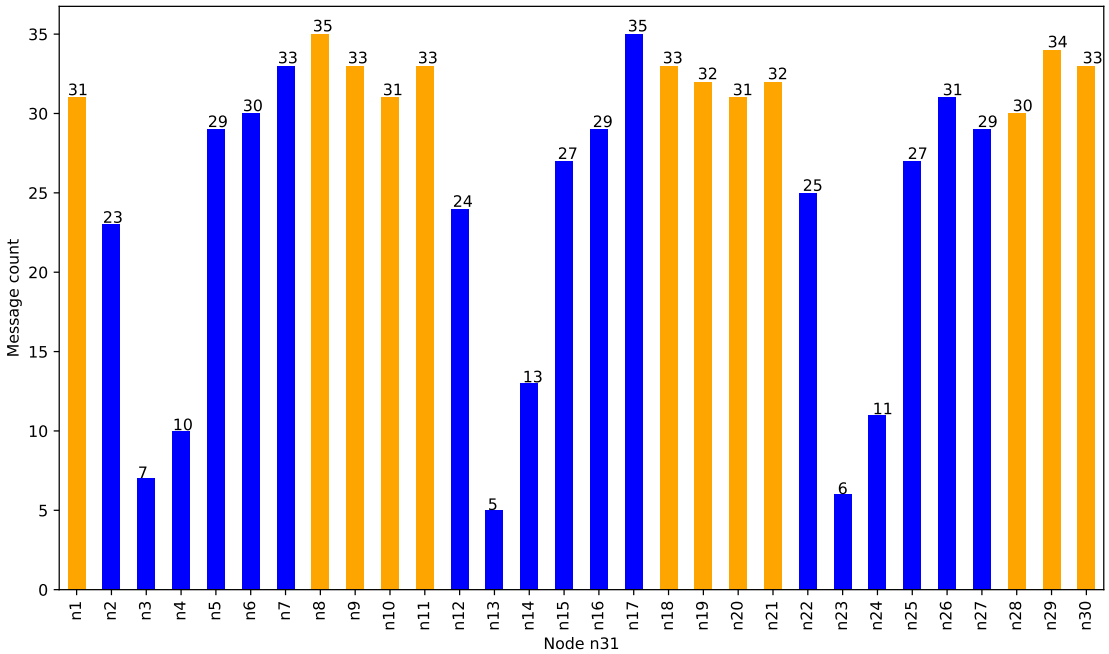


Fig. 4.11: Mensajes recibidos por nodo del *relay* n31 en una ventana de un minuto, marcados con color azul aquellos no conectados y con naranja los conectados.

sender *relay*	account	message count
n40	**n30**	11728
n40	**n9**	11656
n40	**n20**	11634
n40	**n19**	11566
n40	**n10**	11434
n40	**n28**	11424
n40	**n8**	11377
n40	**n18**	11253
n40	**n29**	11024
n40	n21	9050
n40	n12	8793
n40	n2	8788
n40	n22	8766
n40	n1	8721
n40	n11	8663
n40	n17	7012
n40	n7	6885
n40	n27	6860
n40	n6	6845
n40	n16	6777
n40	n26	6379
n40	n13	5528
n40	n3	5423
n40	n14	5242
n40	n24	5136
n40	n23	4973
n40	n4	4472
n40	n15	3870
n40	n5	3726
n40	n25	3564

Tab. 4.2: Cantidad de mensajes por cuenta y *relay* recibidos en el *relay* n35, para el sistema de 30 nodos y 10 *relays* con *stake* homogéneo y tres conexiones lógicas por *relay*.

Sistema:

- **Nodos:** 30 (n1 a n30)
- ***Relays*:** 10 (n31 a n40)
- ***Relays* conectados por nodo:** 2 lógicos, 2 físicos
- ***Relays* conectados por *relay*:** 4 lógicos, 4 físicos
- ***Stake*:** homogéneo
- **Tiempo de ejecución:** 6 h
- **Servidor:** local (1 servidor)

Nuevamente, analizando la tabla 4.3 y la figura 4.13, vemos un decremento en la cantidad de nodos conectados al *relay* elegido (n31), pero que es acorde a la configuración de dos *relays* por nodo: contamos con seis nodos conectados en este *relay* y que mantienen la mayoría distinguida de mensajes recibidos.

Finalmente, realizamos una prueba con un *relay* por nodo:
Sistema:

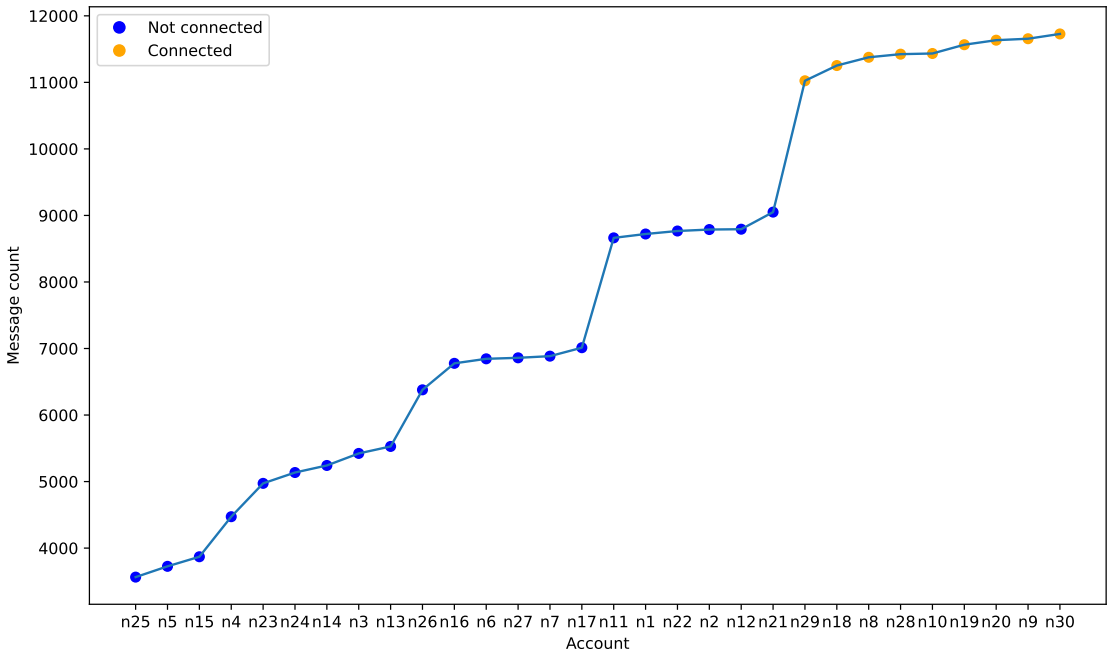


Fig. 4.12: Mensajes recibidos por nodo del *relay* n40 en todo el experimento, marcados con color azul aquellos no conectados y con naranja los conectados

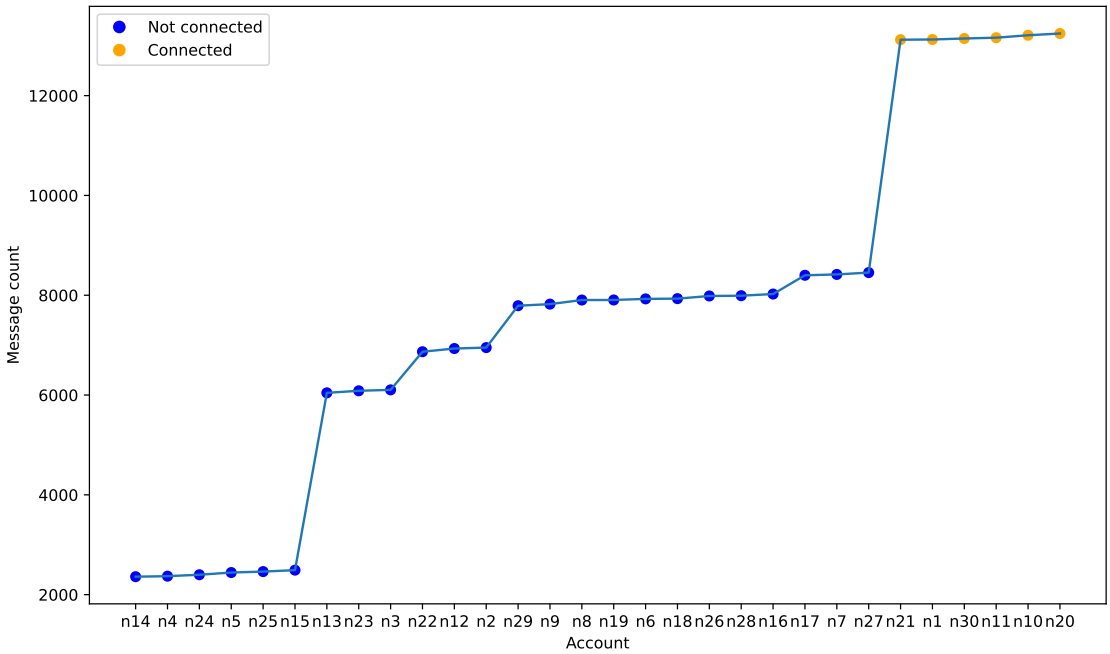


Fig. 4.13: Mensajes recibidos por nodo del *relay* n31 en todo el experimento, marcados con color azul aquellos no conectados y con naranja los conectados.

sender *relay*	account	message count
n31	**n20**	13245
n31	**n10**	13210
n31	**n11**	13161
n31	**n30**	13145
n31	**n1**	13125
n31	**n21**	13121
n31	n27	8454
n31	n7	8417
n31	n17	8400
n31	n16	8024
n31	n28	7991
n31	n26	7985
n31	n18	7932
n31	n6	7927
n31	n19	7905
n31	n8	7904
n31	n9	7822
n31	n29	7790
n31	n2	6951
n31	n12	6932
n31	n22	6868
n31	n3	6104
n31	n23	6085
n31	n13	6044
n31	n15	2491
n31	n25	2462
n31	n5	2443
n31	n24	2399
n31	n4	2370
n31	n14	2360

Tab. 4.3: Cantidad de mensajes por cuenta y *relay* recibidos en el *relay* n35, para el sistema de 30 nodos y 10 *relays* con *stake* homogéneo y dos conexiones lógicas por *relay*.

- **Nodos:** 30 (n1 a n30)
- ***Relays*:** 10 (n31 a n40)
- ***Relays* conectados por nodo:** 1 lógico, 1 físico
- ***Relays* conectados por *relay*:** 4 lógicos, 4 físicos
- ***Stake*:** homogéneo
- **Tiempo de ejecución:** 6 h
- **Servidor:** local (1 servidor)

Siguiendo la tendencia, en base a los datos de la tabla 4.4 y la figura 4.14 contamos con tres nodos conectados al *relay* que conservan las mayores cantidades de mensajes recibidos del mismo.

En base a estos resultados, entendemos que dada cualquier configuración de cantidad de *relays* destino para cada nodo, encontramos una diferenciación notoria entre la cantidad de

sender *relay*	account	message count
n36	**n16**	13266
n36	**n26**	13256
n36	**n6**	13222
n36	n22	11822
n36	n2	11811
n36	n12	11795
n36	n24	9312
n36	n14	9288
n36	n4	9278
n36	n7	8548
n36	n27	8535
n36	n17	8524
n36	n19	8235
n36	n9	8235
n36	n29	8215
n36	n10	7862
n36	n20	7844
n36	n30	7823
n36	n23	7593
n36	n13	7570
n36	n3	7553
n36	n21	7054
n36	n28	7050
n36	n11	7015
n36	n18	6994
n36	n1	6967
n36	n8	6935
n36	n25	1512
n36	n5	1487
n36	n15	1486

Tab. 4.4: Cantidad de mensajes por cuenta y *relay* recibidos en el *relay* n35, para el sistema de 30 nodos y 10 *relays* con *stake* homogéneo y una conexión lógica por *relay*.

mensajes enviados por el mismo según si los nodos se encuentran directamente conectados a él o no. Este salto en la cantidad de mensajes para los conectados versus los no conectados nos da un indicio de que podríamos construir un criterio de corte para poder determinar si un nodo se encuentra conectado o no a un *relay* sin conocer esa información previamente.

A su vez, también sabemos que esta propiedad la observamos sea cual sea la configuración de *relays* por nodos elegida para el sistema. Así, podemos establecer una configuración para experimentar y asumir que los demás casos son análogos al planteado. Para este parámetro elegiremos conectar cada nodo a cuatro *relays* del sistema, ya que ese es la cantidad buscada por cada nodo en la red real.

4.4.3. Análisis por ventana

En la sección anterior tomamos los datos de la corrida completa del experimento para realizar el análisis de los mensajes recibidos y así sacar conclusiones sobre los nodos conectados a cada *relay*. Si bien este enfoque es válido, nuestro interés es simular un ataque a un *relay* que obtiene información de manera sigilosa sin levantar alarmas en el sistema. Para esto, no queremos tener que guardar y enviar información de grandes períodos de tiempo para su análisis, sino que nos interesa encontrar la menor cantidad de ventanas de tiempo de un cierto tamaño que necesitamos para tener una buena aproximación.

Con el fin de poder medir esto, podemos pensar el esfuerzo de obtener esta información relacionando el tamaño de una ventana con la cantidad de ventanas necesarias para obtener un cierto porcentaje de aciertos. Sabemos que cuanto menor es una ventana de tiempo, menor cantidad de información tendrá y es probable que necesitemos más de ellas para llegar a un buen resultado. Pero cuanto mayor sea cada ventana, implicará mayor trabajo extraerla del *relay* sigilosamente y evitar ser detectados. Luego, debemos encontrar un *trade-off* entre tamaño de ventana y cantidad de ventanas necesarias para llegar a un buen acierto.

Otro punto a tener en cuenta es la separación temporal entre ventanas: dado que potencialmente el sistema podría ir mutando, tomar muchas ventanas en un cierto período del experimento puede generar que los resultados difieran con respecto a tomar ventanas distribuidas a lo largo de todo el experimento.

Relacionado con esto último, también podría suceder que en ciertas ventanas los resultados obtenidos no sean los que esperamos: el sistema tiende a ser equitativo con la distribución de las responsabilidades entre todo el *stake* participante del consenso en el tiempo, pero podría suceder que en períodos acotados de tiempo eso no suceda y veamos

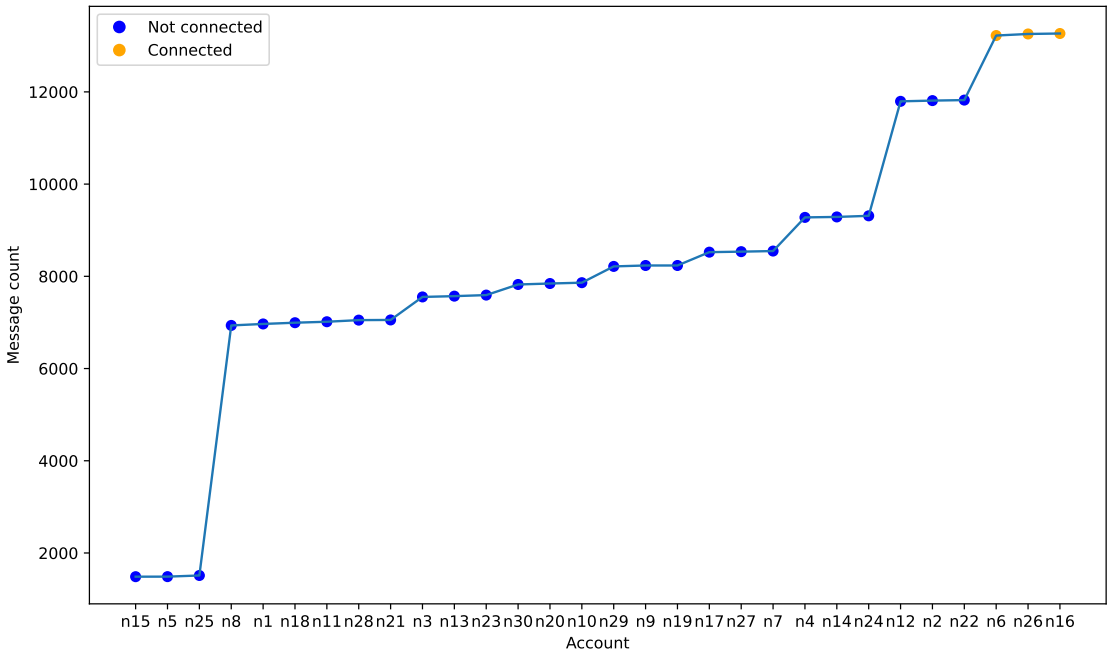


Fig. 4.14: Mensajes recibidos por nodo del *relay* n36 en todo el experimento, marcados con color azul aquellos no conectados y con naranja los conectados.

nodos participando más de lo que les correspondería. Por ejemplo, podría suceder que si todos los nodos tienen la misma probabilidad de crear un bloque, sea el mismo nodo que en rondas seguidas lo cree. La probabilidad de esto debería ser baja según el sistema, pero si llegara a suceder y nosotros tomáramos una sola ventana de tiempo con ese caso, nuestro resultado no sería representativo de toda la corrida. En nuestra línea, esto se podría ver en los casos donde un *relay* nos envía más mensajes de uno de los nodos no conectados a él versus a los conectados.

Entonces, también tenemos que considerar estas aleatoriedades al momento de elegir qué tamaño y cantidad de ventanas vamos a querer tomar para disminuir este ruido y obtener mejores resultados.

En resumen, a menor tamaño de ventana, podemos extraer más sigilosamente información, pero la cantidad y calidad de ella va a ser menor, debiendo tomar mayor cantidad. A su vez, si en la corrida contamos con períodos aislados donde la red presenta alguna anormalidad y nosotros tomamos un período relativamente grande de tiempo para la ventana que lo incluye, estaremos considerando datos que pueden afectar nuestro análisis, mientras que considerando ventanas más pequeñas podríamos aplicar alguna estrategia para que esa ventana anormal no nos afecte considerablemente.

Con estas consideraciones, para el análisis de las ventanas queremos aplicar una metodología que nos permita evitar este ruido y lograr consolidar la información de todas las ventanas muestreadas de manera independiente. Como primer enfoque desglosamos el experimento en ventanas de un tiempo dado, calculamos la cantidad de mensajes recibidos de cada *relay* y la cuenta a la que hace referencia cada uno. Luego, para cada ventana contamos con un listado de cantidad de mensajes por cada cuenta (como los vistos previamente). Utilizando la información de todas las ventanas, podemos sumar la cantidad de mensajes de cada nodo y quedarnos con aquellos que lideran el ranking de mensajes.

A diferencia de lo visto previamente, donde consideramos la cantidad de mensajes totales del experimento, aquí elegimos ventanas de muestreo y sumamos luego los mensajes recibidos en ellas.

El problema de este enfoque es que, si bien vimos que la mayor cantidad de los mensajes enviados por un *relay* están relacionados a las cuentas conectadas directamente a él, puede suceder que en ciertas ventanas no se cumpla estrictamente esto y que hayan muchos mensajes de otra cuenta que no está conectada. Luego, si nosotros sumamos los mensajes de todas las ventanas y no tenemos un muestreo suficiente, puede suceder que este nodo quede en una posición alta en nuestro ranking, cuando la realidad es que solo en una de las ventanas envió muchos mensajes. Entonces, debemos encontrar otra manera de analizar las ventanas que no sea propensa a este tipo de problemas.

En el caso de la tabla 4.5 se puede ver este problema, en una corrida de la misma red utilizada para los anteriores experimentos (30 nodos y 10 *relays*), tomando ventanas de dos minutos para toda la duración del mismo.

Vemos en esta tabla que al ordenar por cantidad de mensajes recibidos, hay algunas cuentas no conectadas que aparecen entre las conectadas o muy cercanas a las mismas.

4.4.4. Puntuando nodos

Dado que la estrategia de solo tomar la cantidad de mensajes es sensible a eventos aislados donde algún *relay* envía más información sobre un nodo que no está conectado a él, buscamos una manera de analizar cada ventana que evite que estos casos afecten el resultado final. Con este fin, en cada ventana ordenamos los nodos por cantidad de

sender *relay*	account	message count
n31	**n1**	2341
n31	**n11**	2318
n31	**n21**	2266
n31	n22	2175
n31	n2	2168
n31	**n29**	2159
n31	**n19**	2136
n31	**n20**	2104
n31	n12	2061
n31	**n10**	2057
n31	**n30**	1923
n31	**n9**	1847
n31	n3	1549
n31	n13	1537
n31	n14	1499
n31	n23	1477
n31	n18	1340
n31	n8	1313
n31	n4	1269
n31	n28	1252
n31	n24	1215
n31	n7	1136
n31	n6	1095
n31	n26	1089
n31	n17	1062
n31	n16	984
n31	n27	966
n31	n5	445
n31	n15	379
n31	n25	376

Tab. 4.5: Cantidad de mensajes por cuenta y *relay* recibidos en el *relay* n35, para el sistema de 30 nodos y 10 *relays* con *stake* homogéneo, analizado por ventanas de 2 min para la totalidad del experimento.

mensajes recibidos, le otorgamos un puntaje a cada uno siguiendo la sucesión de Fibonacci de manera que los nodos con menor cantidad de mensajes reciban un valor bajo y que vaya incrementando según la cantidad de mensajes. Luego, el nodo con mayor cantidad recibirá el valor más alto, teniendo un ranking de nodos siguiendo esta distribución y no una cantidad de mensajes. Finalmente, sumamos los valores de cada ventana para estos pesos asignados y obtenemos una manera de medir la posición de todos los nodos en las distintas ventanas que es mucho menos susceptible a variaciones puntuales en cada una de ellas: básicamente nos desprendemos de contar cantidad de mensajes y nos preocupamos por la posición en el ranking de cada nodo.

Si bien en alguna ventana puede que un nodo no conectado tenga un puntaje alto, debería perder contra los demás nodos que sí están conectados y que ocupan las posiciones

más altas en el ranking en la mayoría de las ventanas.

Aplicamos esta estragtegia de ponderar los mensajes sobre el mismo ejemplo mostrado previamente y obtenemos un ranking con los nodos conectados en las primeras posiciones y con un puntaje mucho mayor que los no conectados, representados en la tabla 4.6.

sender *relay*	**account**	**weight**	**message count**
n31	**n19**	3356472	2136
n31	**n30**	3262403	1923
n31	**n9**	3216986	1847
n31	**n29**	3138493	2159
n31	**n11**	2824325	2318
n31	**n10**	2271639	2057
n31	**n20**	1994030	2104
n31	**n1**	1855241	2341
n31	**n21**	1846805	2266
n31	n8	570502	1313
n31	n7	549828	1136
n31	n22	358984	2175
n31	n12	317593	2061
n31	n2	305085	2168
n31	n6	275099	1095
n31	n13	256550	1537
n31	n4	255739	1269
n31	n3	247179	1549
n31	n23	216067	1477
n31	n27	175693	966
n31	n24	175678	1215
n31	n18	132899	1340
n31	n26	107428	1089
n31	n14	101089	1499
n31	n17	66232	1062
n31	n16	40739	984
n31	n28	23492	1252
n31	n25	2440	376
n31	n5	1893	445
n31	n15	412	379

Tab. 4.6: Cantidad de mensajes y peso asociado por cuenta y *relay* recibidos en el *relay* n35, para el sistema de 30 nodos y 10 *relays* con *stake* homogéneo, analizado por ventanas de 2 min para la totalidad del experimento.

Resulta clave por un lado la mejora que implica agregar pesos en los nodos dentro de cada ventana, no solo porque logra dejar a los conectados primeros, sino también porque marca una clara diferencia entre el peso de los conectados versus los que no lo están. Este resultado nos permite comenzar a elaborar una estrategia para intentar predecir cuáles son los nodos conectados al *relay* basándonos solamente en el peso de cada uno para las ventanas muestreadas.

4.4.5. Estrategia para detectar conectados

Nuestra estrategia para detectar conectados se va a basar en poder identificar el salto del peso entre los nodos no conectados y los conectados al ordenarlos por peso ascendente. Si bien analizándolo a simple vista uno podría deducir este punto de quiebre (al menos en el caso anterior), queremos utilizar un algoritmo que lo automatice y nos permita medir el nivel de éxito de esta distinción.

Como primer estrategia, intentamos detectar ese salto comparando con el tamaño de los demás intervalos: el salto más grande debería ser ese quiebre entre conectados y no conectados. Si bien en algunos casos se cumplía esta relación, vimos que en otros podía suceder que hayan saltos posteriores que sean mayores al de la separación. Por lo tanto, este enfoque no fue exitoso.

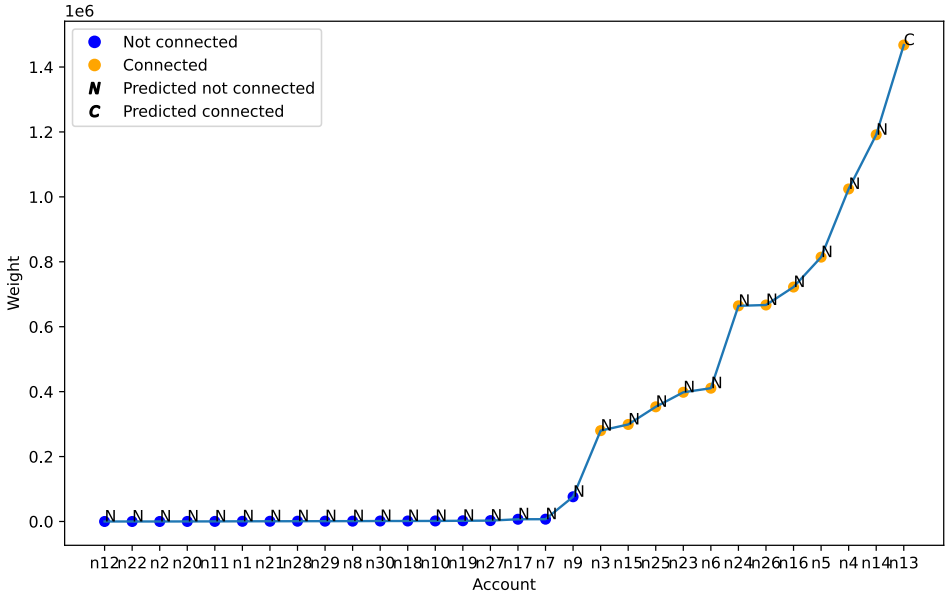


Fig. 4.15: Ejemplo de peso por cuenta para un *relay* con estrategia de predicción para salto más grande, marcados con color azul aquellos no conectados, con naranja los conectados, con una N los predichos como no conectados y una C los predichos como conectados. Sólo el nodo n13 se detecta como conectado al ser el tramo entre n14 y n13 el más grande.

Por ejemplo, en el gráfico 4.15 aplicamos la estrategia previa para detectar los nodos conectados, solo prediciendo al nodo n13 ya que la mayor diferencia entre valores se da en el último intervalo, entre n14 y n13 (superando todas las demás distancias).

En segunda instancia, planteamos analizar los intervalos contra todos los anteriores y suponer que estamos en el salto cuando el incremento en ese intervalo es mayor que el promedio de los saltos anteriores más un cierto porcentaje. Nuevamente, la estrategia funciona en algunos casos, pero en otros donde hay algunos nodos no conectados que envían más mensajes que los anteriores, generaban falsos positivos cuando ese aumento era mayor al porcentaje elegido. La gran falencia de esta estrategia es que no estamos utilizando la información que nos brindan los saltos de los nodos posteriores: quizás entre dos nodos no conectados vemos un incremento de un porcentaje considerable en sus pesos,

pero que a comparación de saltos posteriores no resulta importante.

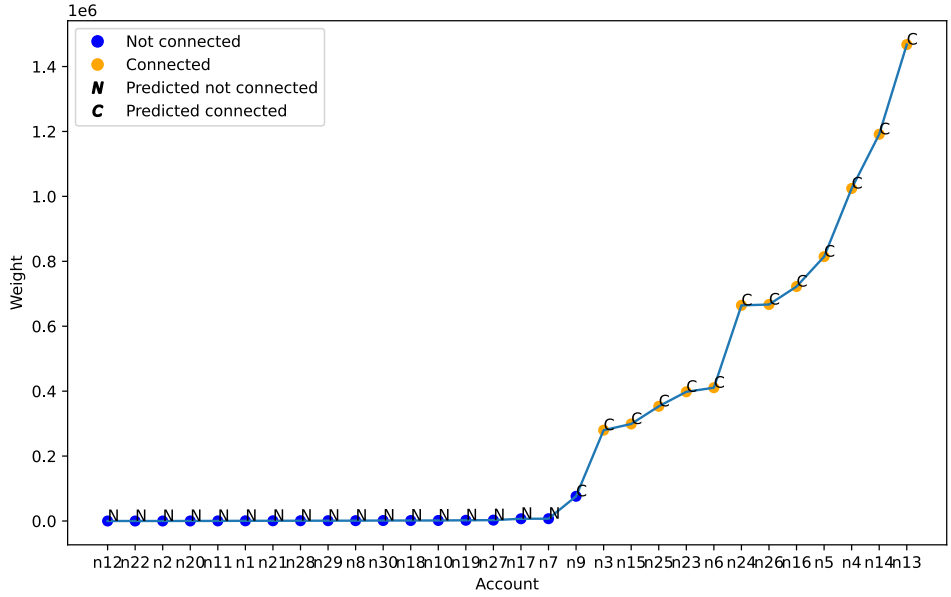


Fig. 4.16: Ejemplo de peso por cuenta para un *relay* con estrategia de predicción donde consideramos el promedio de las distancias anteriores más un porcentaje de desvío del 50 %, marcados con color azul aquellos no conectados, con naranja los conectados, con una N los predichos como no conectados y una C los predichos como conectados. Predecimos el nodo n9 como conectado erróneamente ya que el peso del mismo supera el promedio de todos los anteriores más un 50 %.

En el ejemplo de la imagen 4.16, aplicamos esta última estrategia y si bien predecimos correctamente a todos los conectados, también incluimos al nodo n9 entre ellos cuando no debería estarlo. El problema en este caso es que el nodo n9 presenta una cantidad mayor de mensajes en comparación con los nodos anteriores, pero que resulta sumamente inferior que los siguientes.

Teniendo esto en cuenta, generamos una nueva estrategia que combina lo aprendido de las anteriores. En primer lugar, el salto que buscamos debería ser mayor que el promedio de todos los saltos anteriores más un porcentaje seteado en un 50 %. Y, en segundo, debería ser mayor que el promedio de los saltos posteriores, sin agregarle un incremento porcentual, ya que solo queremos que este salto sea mayor o similar a los intervalos siguientes. Con estos dos criterios, vamos a buscar el punto de quiebre donde se cumplen ambas propiedades a la vez.

En el caso de la figura 4.17, la estrategia funciona correctamente y no incluimos al nodo n9 dentro de los predichos como conectados ya que su incremento resulta menor que el promedio de los posteriores. Así, logramos predecir de buena manera los conectados y los no conectados.

Todas estas estrategias nos van a dividir el conjunto de los nodos de los cuales recibimos mensajes desde un *relay* en dos grupos: potencialmente conectados y no conectados. Por supuesto que este algoritmo puede fallar en casos donde la diferencia de los intervalos no sea tan notoria o si hay comportamientos que no siguen lo esperado para la distribución

de mensajes. Nos importa más que nada poder predecir con precisión cuáles son los nodos conectados efectivamente al *relay*, aunque no los detectemos a todos.

Llevando esto último a términos de análisis de datos, vamos a medir la precisión y el *recall* de cada experiencia con respecto a la predicción de nodos conectados. La precisión hace referencia a qué porcentaje de los nodos identificados como conectados están efectivamente conectados (es decir, verdaderos positivos), según la fórmula 4.1.

Mientras que el *recall* nos da la proporción de nodos predichos como conectados sobre todos los efectivamente conectados (es decir, qué cantidad de nodos conectados pudimos detectar), expresado en la fórmula 4.2. Idealmente queremos que ambos valores se acerquen lo más posible al 100 %, donde predecimos como conectados a todos los nodos que están realmente conectados y no perdemos ninguno de ese conjunto, pero con la menor cantidad de ventanas posibles.

$$Precisión = \frac{\#\, Nodos\ conectados\ con\ predicción\ positiva}{\#\, Nodos\ con\ predicción\ positiva} \tag{4.1}$$

$$Recall = \frac{\#\, Nodos\ conectados\ con\ predicción\ positiva}{\#\, Nodos\ conectados} \tag{4.2}$$

Recordando lo explicado previamente, queremos que este análisis pase desapercibido por el *relay* infiltrado, con lo cual nos interesa mucho encontrar un buen *trade-off* entre buena precisión, buen *recall* y baja cantidad de ventanas tomadas.

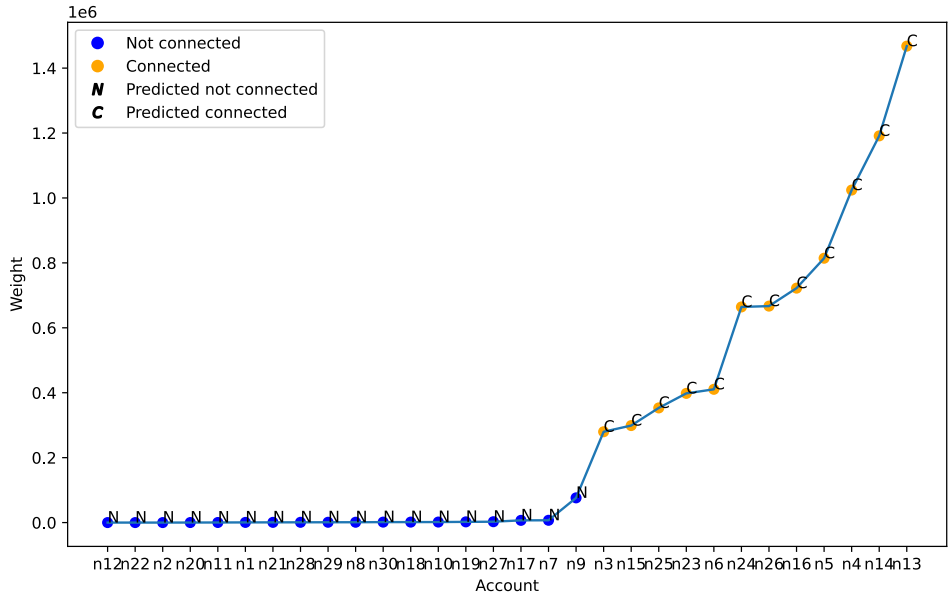


Fig. 4.17: Ejemplo de peso por cuenta para un *relay* con estrategia de predicción que considera los valores anteriores y posteriores, marcados con color azul aquellos no conectados, con naranja los conectados, con una N los predichos como no conectados y una C los predichos como conectados. Predecimos correctamente todos los nodos conectados y no conectados del sistema.

4.4.6. Selección de ventanas

Como describimos previamente, es sumamente importante la manera en la que se seleccionan las ventanas durante el experimento para poder analizar la información que nos brinda cada una. Para aprovechar la duración completa de cada experimento, vamos a definir el tamaño de las ventanas a tomar primero y luego, en base a la cantidad que queremos tomar, distribuirlas semi-equidistantemente (buscando una separación equitativa entre ellas, más un porcentaje mínimo de variabilidad, que rondará entre el 0 % a 5 % de su duración).

De esta manera, nos aseguramos de tomar ventanas espaciadas e intentar contemplar todos los escenarios que plantea el experimento durante su duración, evitando tomar ventanas muy cercanas que pudieran sesgar nuestras observaciones por alteraciones temporales.

Para toda la experimentación que continúa se aplica esta estrategia de selección, aclarando debidamente si no fuera de esa manera.

4.5. Resumen de los experimentos y resultados

- Dividimos la experimentación en dos secciones, una con distribución homogénea del *stake*, y otra con distribución heterogénea que respeta la vista en la red real de `Algorand`.
- Utilizamos cuatro tamaños de red incrementales: 30 nodos y 10 *relays* (pequeño), 60 nodos y 20 *relays* (mediano), 150 nodos y 50 *relays* (semi-realista), y 300 nodos y 100 *relays* (realista).
- En cada red las conexiones físicas y lógicas tanto de nodos como de *relays* se mantienen fijas:
  - Los nodos se conectan física y lógicamente a cuatro *relays*.
  - Los *relays* se conectan lógicamente a otros cuatro, y en físicamente en hasta tres saltos a los demás.
- Analizamos los datos con distintas cantidades y tamaños de ventanas, utilizando un algoritmo de predicción base para distinguir cuentas conectadas de no conectadas.
- Medimos la precisión, *recall* y esfuerzo de las predicciones para cada combinación.

4.5.1. Escenarios de *stake* homogéneo

- 30 nodos y 10 *relays* (pequeño)
  - **Mejor resultado:** 100 % de precisión y 100 % de *recall* con 50 ventanas de un minuto (esfuerzo de 50 min).
- 60 nodos y 20 *relays* (mediano)
  - **Mejor resultado:** 100 % de precisión y 100 % de *recall* con diez ventanas de dos minutos (esfuerzo de 20 min).
- 150 nodos y 50 *relays* (semi-realista)
  - **Mejor resultado:** 100 % de precisión y 100 % de *recall* con 25 ventanas de cinco minutos (esfuerzo de 125 min).

- 300 nodos y 100 *relays* (realista)
  - El gran tamaño del sistema impacta negativamente en las predicciones.
  - **Mejor resultado:** 92 % de precisión y 44 % de *recall* con diez ventanas de diez minutos (esfuerzo de 100 min).

4.5.2. Escenarios de *stake* heterogéneo

- El algoritmo de predicción anterior (**base**) falla notoriamente al no considerar la distribución heterogénea del *stake*.
- Implementamos dos algoritmos nuevos que lo consideran, ponderando las mediciones con el *stake* de cada cuenta: **count** y **fibo**.
- **fibo** logra mejores resultados en todos los escenarios, mejorando sustancialmente el algoritmo **base**.
- Creamos tres nuevas variantes dentro de cada tamaño de red según la distribución de los nodos entre los *relays*:
  - **Semi-equitativo**: cada *relay* maneja un promedio similar de *stake*, con un 20 % de desvío.
  - ***Stake* cercano**: se conectan las cuentas de mayor *stake* a los *relays* más cercanos al *relay* analizado, progresivamente disminuyendo su *stake* a medida que se alejan.
  - ***Stake* lejano**: se conectan las cuentas de menor *stake* a los *relays* más cercanos al *relay* analizado, progresivamente aumentando su *stake* a medida que se alejan.
- Creamos un nuevo experimento para medir la detección del **Top 5** de cuentas según *stake*, con un escenario donde se encuentran conectadas a los *relays* vecinos del *relay* analizado, y otro donde no, con escala realista.
  - Aquí la precisión y *recall* hacen referencia a la detección de la conexión (o no) de estas cinco cuentas a los *relays vecinos*.

En base a esto, obtenemos los siguientes resultados para la mejor estrategia, **fibo**:

- 30 nodos y 10 *relays* (pequeño)
  - **Semi-equitativo - Mejor resultado:** 100 % de precisión y 98 % de *recall* con 200 ventanas de un minuto (esfuerzo de 200 min).
  - ***Stake* cercano - Mejor resultado:** 100 % de precisión y 100 % de *recall* con diez ventanas de un minuto (esfuerzo de 10 min).
  - ***Stake* lejano - Mejor resultado:** 72 % de precisión y 85 % de *recall* con diez ventanas de un minuto (esfuerzo de 10 min).
- 60 nodos y 20 *relays* (mediano)
  - **Semi-equitativo - Mejor resultado:** 98 % de precisión y 96 % de *recall* con 300 ventanas de un minuto (esfuerzo de 300 min).
  - ***Stake* cercano - Mejor resultado:** 100 % de precisión y 100 % de *recall* con 50 ventanas de dos minutos (esfuerzo de 100 min).

  - ***Stake* lejano - Mejor resultado:** 89 % de precisión y 93 % de *recall* con 25 ventanas de un minuto (esfuerzo de 25 min).

- 150 nodos y 50 *relays* (semi-realista)
  - **Semi-equitativo - Mejor resultado:** 100 % de precisión y 44 % de *recall* con 25 ventanas de dos minutos (esfuerzo de 50 min).
  - ***Stake* cercano - Mejor resultado:** 95 % de precisión y 58 % de *recall* con 25 ventanas de cinco minutos (esfuerzo de 125 min).
  - ***Stake* lejano - Mejor resultado:** 72 % de precisión y 68 % de *recall* con diez ventanas de diez minutos (esfuerzo de 100 min).

- 300 nodos y 100 *relays* (realista)
  - **Semi-equitativo - Mejor resultado:** 93 % de precisión y 19 % de *recall* con 100 ventanas de cinco minutos (esfuerzo de 500 min).
  - ***Stake* cercano - Mejor resultado:** 82 % de precisión y 100 % de *recall* con 25 ventanas de diez minutos (esfuerzo de 250 min).
  - ***Stake* lejano - Mejor resultado:** 16 % de precisión y 83 % de *recall* con 25 ventanas de diez minutos (esfuerzo de 250 min).

- 300 nodos y 100 *relays* (realista): reconociendo el **Top 5**
  - **Top 5 conectado - Mejor resultado:** 100 % de precisión y 100 % de *recall* con 25 ventanas de un minuto (esfuerzo de 25 min).
  - **Top 5 no conectado - Mejor resultado:** 100 % de precisión y 100 % de *recall* con cinco ventanas de un minuto (esfuerzo de 5 min).

4.6. Experimentación con *stake* homogéneo

4.6.1. Caso pequeño

Aplicamos esta estrategia para un sistema de 30 nodos y 10 *relays* inicialmente, con *stake* homogéneo, tomando ventanas distribuidas semi-equitativamente a lo largo de la duración del experimento y variando su cantidad y tamaño. Para cada una de ellas calculamos la precisión y *recall* de detección de conectados con la estrategia descrita previamente.

Sistema:

- **Nodos:** 30 (n1 a n30)
- ***Relays*:** 10 (n31 a n40)
- ***Relays* conectados por nodo:** cuatro lógicos, cuatro físicos (con latencia de 20 ms)
- ***Relays* conectados por *relay*:** cuatro lógicos, cuatro físicos (con latencia de 30 ms)
- ***Stake*:** homogéneo
- **Tiempo de ejecución:** 10 h
- **Servidor:** local (un servidor)

Siguiendo los resultados de la tabla 4.7, la columna “window size” hace referencia al tamaño de ventana utilizado en minutos, “window count” la cantidad de ventanas de ese tamaño tomadas de todo el experimento, y “total mins” al total de minutos de muestras que representa esta combinación de tamaño y cantidad. Vemos que para ventanas de 1 minuto, logramos una precisión del 100 % ya desde el comienzo, pero con un *recall* bajo para el caso de cinco muestras. A medida que aumentamos la cantidad de ventanas tomadas, este valor aumenta notoriamente llegando al 100 % ya con 100 ventanas muestreadas. Luego

window size (mins)	window count	total mins	connected precision	connected *recall*
1	5	5	1	0.6852
1	10	10	1	0.713
1	25	25	0.9778	0.9537
1	50	50	1	1
1	100	100	1	1
1	150	150	1	1
1	200	200	1	1
1	250	250	1	1
1	300	300	1	1
1	350	350	1	1
2	5	10	1	0.6389
2	10	20	1	0.9259
2	25	50	1	0.9722
2	50	100	0.9915	1
2	100	200	1	1
2	150	300	1	1
5	5	25	1	0.6019
5	10	50	0.9915	0.963
5	25	125	0.9915	1
5	50	250	0.9915	1
10	5	50	1	0.5926
10	10	100	1	0.9722
10	25	250	0.9915	1
20	5	100	1	0.7685
20	10	200	0.9915	0.9907
30	5	150	1	0.75
30	10	300	1	0.9907

Tab. 4.7: Precisión y *recall* según cantidad y tamaño de ventana para la predicción de cuentas conectadas a los *relays* asociados al *relay* n35, para el sistema de 30 nodos y 10 *relays* con *stake* homogéneo.

se mantiene el resultado al aumentar la cantidad de ventanas, logrando una detección perfecta de los nodos conectados al *relay*.

En los demás tamaños de ventanas sucede lo mismo: si bien a medida que aumentamos el tamaño de la misma, el *recall* para la misma cantidad de ventanas incrementa, tenemos que tomar más ventanas para que este sea del 100 %.

Por ejemplo, tomamos uno de los *relays* analizados para el caso de cinco ventanas de un minuto y lo representamos en la figura 4.18.

Marcamos con color naranja las cuentas que estan efectivamente conectadas al *relay* analizado, mientras que los azules representan aquellas que no lo están. También, sobre las mismas colocamos una N para aquellas cuentas que nuestro algoritmo predice que no está conectada y una C para las que sí. En este caso, vemos que nuestro algoritmo predice como conectadas las cuentas a partir de n4 hacia la derecha, reconociendo el 50 % de las cuentas efectivamente conectadas al *relay*. De esta manera, nuestra precisión es del 100 %, mientras que el *recall* es del 60 %, ya que si bien todas nuestras predicciones positivas son

verdaderas, nos falta reconocer un 40 % de las cuentas que están conectadas.

A diferencia de esto, vemos lo que sucede al tomar 100 de estas ventanas de un minuto en el caso de la figura 4.19.

Notamos que con mayor información, la diferencia entre nodos conectados y no conectados se marca fuertemente y permite que nuestro algoritmo prediga la totalidad de los nodos conectados correctamente.

Algo interesante para analizar con estos resultados es la comparación entre mismos tiempos de muestreo, pero con diferentes tamaños de ventana: es decir, comparar por ejemplo el resultado de tomar diez ventanas de un minuto versus cinco ventanas de dos minutos. Para esto, podemos utilizar la columna de "total mins" para comparar el esfuerzo de cada combinación. Vemos que 100 ventanas de un minuto logramos precisión y *recall* perfectos, mientras que con ventanas de dos minutos lo logramos con la mitad del esfuerzo: con 25 ventanas logramos el mismo resultado. En el caso de ventanas de cinco minutos, el resultado con diez ventanas es similar al obtenido con 50 de un minuto, con lo cual no mejora el resultado. Finalmente, en las ventanas más grandes el esfuerzo necesario para obtener este resultado es mucho mayor.

A partir de este experimento podemos concluir que con ventanas de dos minutos logramos un resultado muy bueno con el menor esfuerzo, seguido por ventanas de cinco minutos. También corroboramos nuestras hipótesis previas, con ventanas de un minuto es necesario tomar muchas más muestras para poder llegar al mismo resultado, algo que sucede también con ventanas mucho más grandes. En el primer caso, pareciera ser que la información obtenida de cada una es relativamente pequeña, con lo cual necesitamos más

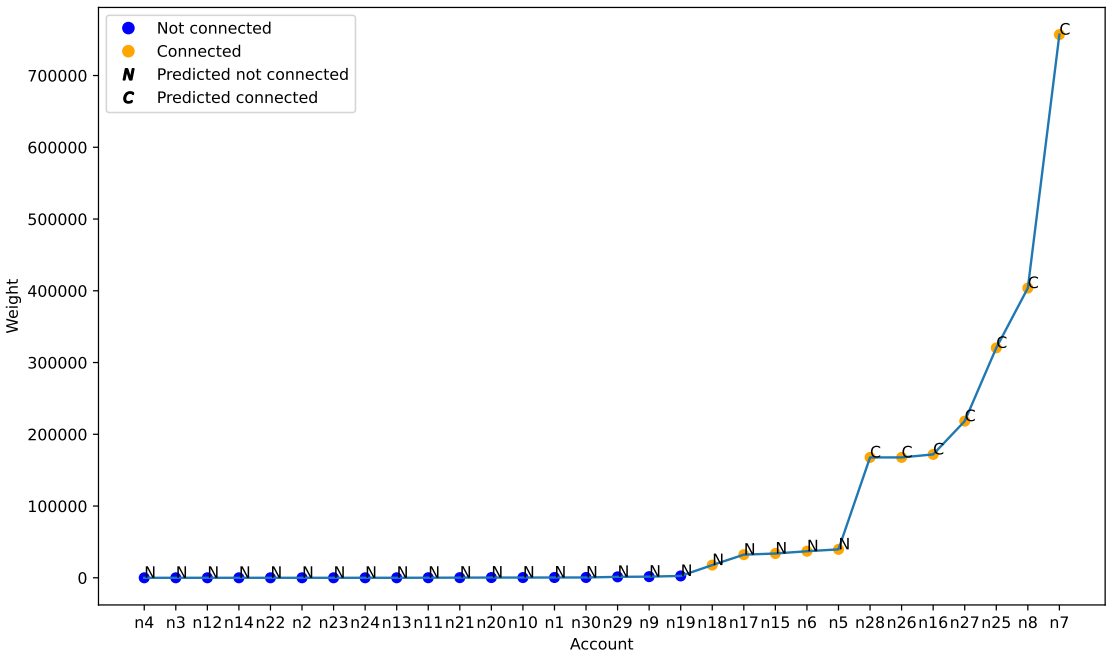


Fig. 4.18: Peso por cuenta para el *relay* n38 tomando cinco ventanas de un minuto cada una, marcados con color azul aquellos no conectados, con naranja los conectados, con una N los predichos como no conectados y una C los predichos como conectados.

ventanas. En el segundo, es probable que como la ventana es grande, estemos considerando mucho ruido en el análisis y nuevamente necesitemos más muestras para disminuirlo.

4.6.2. Caso mediano

Repetimos la experiencia, pero para 60 nodos y 20 *relays*, duplicando el tamaño del experimento anterior para validar lo obtenido aquí. Dado que aumentamos la cantidad de *relays*, también aumentamos la cantidad de enlaces físicos para asegurarnos de mantener la conectividad entre todos los nodos con una latencia similar. La topología física respeta la misma distribución que en el sistema de 30 nodos y 10 *relays*.

Sistema:

- **Nodos:** 60 (n1 a n60)
- ***Relays*:** 20 (n61 a n80)
- ***Relays* conectados por nodo:** 4 lógicos, 4 físicos (con latencia de 20 ms)
- ***Relays* conectados por *relay*:** 4 lógicos, 8 físicos (con latencia de 30 ms)
- ***Stake*:** homogéneo
- **Tiempo de ejecución:** 10 h
- **Servidor:** local (1 servidor)

En este caso, reflejado en la tabla 4.8, el resultado obtenido es muy similar a lo visto en el experimento con 30 nodos y 10 *relays*, vemos en la tabla que el *recall* sube a medida que tomamos más ventanas de cada tamaño, llegando rápidamente al 100 % junto con la precisión, siendo menor para una cantidad baja de ventanas. Ya a partir de la segunda cantidad nos acercamos casi al 100 % de ambas magnitudes en la mayoría de los casos.

Nuevamente, vemos que el mejor resultado en cuanto al esfuerzo lo obtenemos tomando

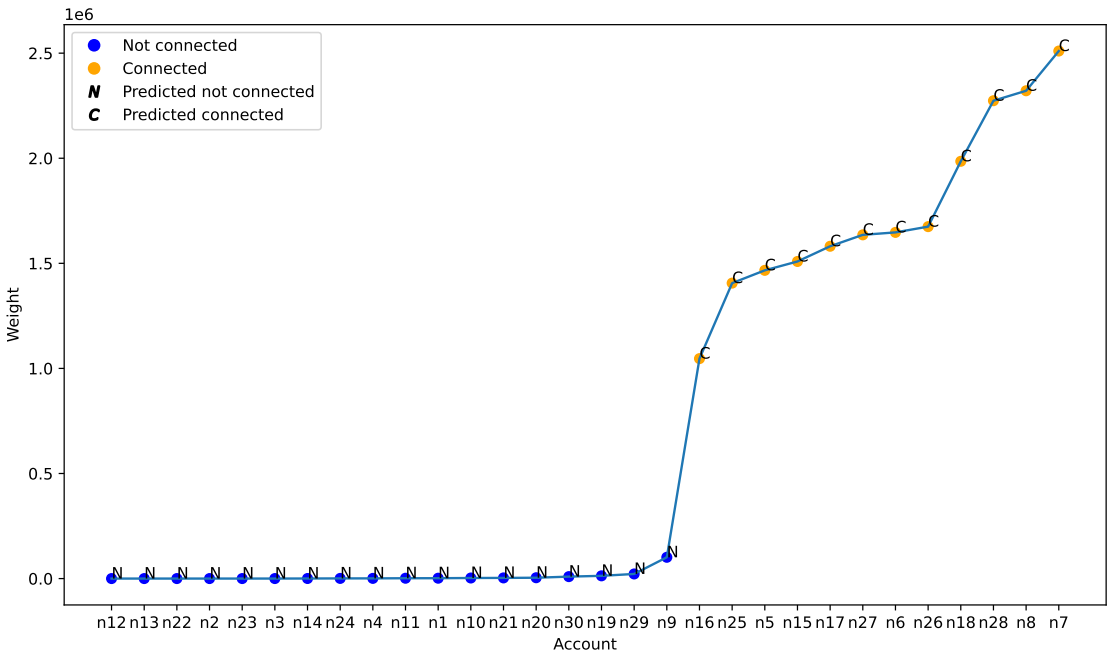


Fig. 4.19: Peso por cuenta para el *relay* n38 tomando 100 ventanas de un minuto cada una, marcados con color azul aquellos no conectados, con naranja los conectados, con una N los predichos como no conectados y una C los predichos como conectados.

window size (mins)	**window count**	**total mins**	**connected precision**	**connected *recall***
1	5	5	0.9615	0.3958
1	10	10	0.9219	0.9792
1	25	25	1	0.9792
1	50	50	0.9808	0.9792
1	100	100	0.95	1
1	150	150	0.95	1
1	200	200	1	1
1	250	250	1	1
1	300	300	1	1
1	350	350	1	1
1	400	400	1	1
1	450	450	1	1
1	500	500	1	1
2	5	10	1	0.5833
2	10	20	1	1
2	25	50	1	1
2	50	100	1	1
2	100	200	1	1
2	150	300	1	1
2	200	400	1	1
2	250	500	1	1
5	5	25	1	0.75
5	10	50	1	1
5	25	125	1	1
5	50	250	1	1
5	100	500	1	1
10	5	50	1	0.8125
10	10	100	1	0.9583
10	25	250	1	1
10	50	500	1	1
20	5	100	1	0.8333
20	10	200	1	0.9792
20	25	500	1	1
30	5	150	1	0.9167
30	10	300	1	0.9792

Tab. 4.8: Precisión y *recall* según cantidad y tamaño de ventana para la predicción de cuentas conectadas a los *relays* asociados al *relay* n65, para el sistema de 60 nodos y 20 *relays* con *stake* homogéneo.

diez ventanas de dos minutos (con precisión y *recall* del 100 %).

A pesar de estas diferencias, cabe destacar que las métricas obtenidas resultan muy optimistas en ambos sistemas, con solo tomar diez ventanas de un minuto ya logramos precisión y *recall* superior al 90 %.

4.6.3. Escalando el experimento: caso semi-realista

Como siguiente paso, escalamos el tamaño de nuestra red para analizar cuál es su impacto en las estrategias de predicción desarrolladas. Nos interesa conocer si se mantiene el mismo comportamiento o si este incremento en la cantidad de nodos y *relays* afecta directamente nuestra predicción.

La red analizada cuenta con 150 nodos y 50 *relays*, con la siguiente estructura:

Sistema:

- **Nodos:** 150 (n1 a n150)
- **Relays:** 50 (n151 a n200)
- **Relays conectados por nodo:** 4 lógicos, 4 físicos (con latencia de 20 ms)
- **Relays conectados por *relay*:** 4 lógicos, 18 físicos (con latencia de 30 ms)
- **Stake:** homogéneo
- **Tiempo de ejecución:** 10 h
- **Servidor:** CloudLab (5 servidores *d6515*)

Nuevamente incrementamos la cantidad de conexiones físicas de cada *relay* para asegurar la conectividad de manera uniforme en todo el sistema, teniendo hasta tres saltos como máximo para conectarse entre ellos.

Se corre el experimento y obtenemos los siguientes resultados:

A partir de los datos de la tabla 4.9, vemos que si bien logramos buenas predicciones, el esfuerzo necesario para obtenerlas crece. Esto tiene sentido ya que la red cuenta con más del doble de nodos que la red anterior, con lo cual cada *relay* va a estar enviando información de 150 nodos y vamos a requerir mayor cantidad de datos para poder hacer una buena predicción.

Es notable como la precisión de las ventanas de uno y dos minutos bajan en su calidad comparado con los casos anteriores. Esto nos da un indicio que para redes más grandes quizás no es una buena idea tomar una ventana de tiempo tan pequeña: obtenemos mejores resultados con ventanas de tiempo más grandes. Por ejemplo, con ventanas de dos minutos requerimos un esfuerzo de 200 minutos para acercarnos al mejor caso de precisión y *recall*, mientras que con ventanas de cinco minutos alcanza con tomar 25 para lograr casi la predicción perfecta (lo cual equivale a 125 minutos de esfuerzo, alrededor de la mitad). Para ventanas de 10, 20 y 30 minutos las predicciones resultan buenas al tomar diez o más ventanas, pero el esfuerzo en esos casos resulta mayor también.

Para verificar esta tendencia, continuamos incrementando la red y buscamos validarla.

4.6.4. Caso realista

Analizamos una red que intenta replicar la arquitectura actual de la real, contando con 300 nodos y 100 *relays* conectados en la misma. La idea es analizar si esta estrategia funcionaría en el caso de que atacáramos un *relay* en la red actual para obtener información.

Sistema:

- **Nodos:** 300 (n1 a n300)
- ***Relays*:** 100 (n301 a n400)
- ***Relays* conectados por nodo:** 4 lógicos, 4 físicos (con latencia de 20 ms)
- ***Relays* conectados por *relay*:** 4 lógicos, 46 físicos (con latencia de 30 ms)
- ***Stake*:** homogéneo
- **Tiempo de ejecución:** 10 h

window size (mins)	window count	total mins	connected precision	connected *recall*
1	5	5	0.8264	0.2292
1	10	10	0.904	0.375
1	25	25	0.9545	0.3542
1	50	50	0.8864	0.625
1	100	100	0.803	0.7083
1	150	150	0.8669	0.9792
1	200	200	0.8875	1
1	250	250	0.8661	1
1	300	300	0.8	1
1	350	350	0.9143	1
1	400	400	0.8661	1
1	450	450	0.9018	1
1	500	500	0.8661	1
2	5	10	0.9233	0.5417
2	10	20	0.8951	0.9375
2	25	50	0.8335	0.9792
2	50	100	0.8615	1
2	100	200	1	0.9792
2	150	300	0.9615	1
2	200	400	0.9079	1
2	250	500	0.9079	1
5	5	25	1	0.4375
5	10	50	0.9808	0.75
5	25	125	1	1
5	50	250	0.9643	1
5	100	500	1	1
10	5	50	1	0.8125
10	10	100	1	0.8958
10	25	250	1	1
10	50	500	1	1
20	5	100	1	0.7083
20	10	200	1	0.9792
20	25	500	1	1
30	5	150	1	0.6667
30	10	300	1	1

Tab. 4.9: Precisión y *recall* según cantidad y tamaño de ventana para la predicción de cuentas conectadas a los *relays* asociados al *relay* n165, para el sistema de 150 nodos y 50 *relays* con *stake* homogéneo.

- **Servidor:** CloudLab (20 servidores *d6515*)

Cabe resaltar que la gran cantidad de enlaces físicos entre los *relays* no responden a características realistas, sino que los utilizamos para que el *delay* entre los mismos sea homogéneo y de hasta tres saltos entre ellos.

Con esta red se obtienen los siguientes resultados:

Analizando los datos obtenidos para el experimento en la tabla 4.10, vemos un decre-

window size (mins)	**window count**	**total mins**	**connected precision**	**connected *recall***
1	5	5	0.2881	0.3056
1	10	10	0.5	0.4167
1	25	25	0.5172	0.5556
1	50	50	0.4889	0.75
1	100	100	0.4617	0.8889
1	150	150	0.5466	0.8889
1	200	200	0.5413	0.8333
1	250	250	0.3284	0.9722
1	300	300	0.7014	0.7222
1	350	350	0.4614	0.9444
1	400	400	0.5101	0.9167
1	450	450	0.4317	0.9444
1	500	500	0.4868	0.9444
1	600	600	0.6037	0.75
2	5	10	0.293	0.3333
2	10	20	0.6556	0.5833
2	25	50	0.8333	0.3333
2	50	100	0.9333	0.1944
2	100	200	0.8041	0.6389
2	150	300	0.504	0.8056
2	200	400	0.4127	0.9444
2	250	500	0.3892	0.9444
5	5	25	0.4694	0.4167
5	10	50	0.7875	0.3889
5	25	125	0.7333	0.6111
5	50	250	0.7222	0.7222
5	100	500	0.8413	0.5
10	5	50	1	0.2222
10	10	100	0.9213	0.4444
10	25	250	0.9091	0.3611
10	50	500	1	0.3056
20	5	100	0.9048	0.3611
20	10	200	0.9524	0.4167
20	25	500	1	0.2778
30	5	150	0.9444	0.3333
30	10	300	1	0.3889

Tab. 4.10: Precisión y *recall* según cantidad y tamaño de ventana para la predicción de cuentas conectadas a los *relays* asociados al *relay* n301, para el sistema de 300 nodos y 100 *relays* con *stake* homogéneo.

mento marcado en las métricas obtenidas para las ventanas más pequeñas, teniendo muy mala precisión, aunque con algunos buenos resultados en el *recall* a medida que aumentan la cantidad de ventanas tomadas. Luego, para ventanas de cinco minutos nuevamente vemos resultados similares que para los otros tamaños menores, pero con ventanas a partir de los diez minutos logramos mejorar fuertemente la precisión de la predicción. Sin embargo,

el *recall* en esos casos se mantiene por debajo del 50 %, sin parecer mejorar al incrementar la cantidad de ventanas tomadas.

Con estos resultados, vemos que al ser un sistema mucho más grande, es más difícil realizar predicciones y como todos los nodos tienen la misma participación en el sistema, no hay ninguno que se destaque por sobre los demás. Como resultado optimista, para las ventanas grandes si bien perdemos muchos nodos conectados, estamos seguros que los que son predichos como conectados lo están (ya que la precisión se mantiene cercana al 100 %). Si diéramos con alguno de los nodos que estamos buscando con estas ventanas y lo predijéramos como conectado, tendríamos mucha confianza de que efectivamente lo está.

Todos estos casos plantean una red en un estado perfecto de homogeneidad, un caso que podría decirse que es el más democrático en una red descentralizada, pero que no responde a la realidad actual ni la que suele darse en estas redes.

4.7. Experimentación con *stake* heterogéneo

4.7.1. Análisis inicial

En el caso anterior, todas las cuentas que participan del consenso lo hacen con el mismo peso: todas tienen la misma cantidad de *stake* del sistema. Luego, la probabilidad de que cada una de ellas sea elegida para formar el siguiente bloque y participar en el protocolo de consenso para ellos es la misma. Como vimos en la introducción, en `Algorand` las cuentas participan del consenso votando según la cantidad de *stake* que posean: a mayor cantidad de *stake*, mayor cantidad de votos. En base a estas dos cuestiones, la cantidad de mensajes enviados por cada cuenta no seguirá siendo uniforme como veíamos en el caso anterior, sino que tendremos una nueva dificultad en el análisis.

Anteriormente, si analizábamos los mensajes recibidos de un *relay*, como todas las cuentas emitían una cantidad similar, ese *relay* siempre debería ser el primero en reenviar los suyos versus los demás (al menos con una cierta cantidad de muestras). En este nuevo escenario, como las distribuciones cambian, podríamos recibir muchos mensajes de una cuenta con mucho *stake* que opaque los mensajes de los nodos efectivamente conectados al *relay*. En el mejor caso, este nodo también va a estar conectado al *relay* y deberíamos poder detectarlo fácilmente, pero en caso contrario va a estar generando incongruencias si seguimos el algoritmo planteado previamente.

Vamos a experimentar entonces con redes con *stake* distribuido de manera no equiforme para simular una red más realista e intentar aplicar algoritmos que nos permitan seguir identificando a las cuentas conectadas a un *relay*.

Nuevamente, cada escenario se repite tres veces para validar los resultados obtenidos, verificando los resultados más allá de los factores aleatorios de cada ejecución.

4.7.2. Reproduciendo la distribución del *stake*

Para comenzar con esta sección decidimos intentar replicar en nuestras redes una distribución del *stake* que se condiga con la distribución en la red actual. Como vimos en la introducción, los datos sobre la distribución actual son públicos y conocemos la cantidad de nodos y sus *stakes* que participan del consenso.

Así, para nuestros experimentos con distribución realista del *stake*, nos basamos en esta gráfica para generar los *stakes* de cada nodo de manera que sea lo más similar a la distribución real.

Más allá de la distribución del *stake* entre los nodos, también queremos que estos nodos se encuentren conectados a *relays* de manera que cada *relay* reciba información de nodos con *stakes* variados. Para esto, en cada experimento diseñamos las conexiones de manera de mantener un promedio de *stake* por *relay* dependiente del tamaño de la red, más un factor de desvío estándar relativo (*RSD*) del 20 % para evitar que se encuentre perfectamente balanceada (y así acercarnos a un escenario más realista).

Para cada sistema, se especifica el valor promedio del *stake* total que manejan los nodos conectados a cada *relay* (indicado como **Promedio de *stake* por *relay***.).

Además de esta distribución, y con el fin de poder comparar los resultados con otros escenarios, planteamos dos sistemas alternativos donde distribuimos el *stake* de la red según su distancia al *relay* analizado: en uno de ellos, concentramos el *stake* lo más cercano posible al *relay*, quedando la mayoría del mismo a la menor cantidad de saltos posibles. Por otro lado, planteamos el escenario contrario, donde alejamos la mayor cantidad de *stake* del *relay* analizado, conectando las cuentas que contienen una mayor cantidad a los nodos que se encuentran a mayor cantidad de saltos.

Estos dos escenarios nos agregan información sobre el comportamiento de la red para distribuciones alternativas del *stake* en la misma.

4.7.3. Caso pequeño con estrategia anterior

Al igual que con la experimentación previa, realizamos un experimento pequeño con 30 nodos y 10 *relays* donde la distribución del *stake* respeta la realista y analizamos los resultados obtenidos.

Sistema:

- **Nodos:** 30 (n1 a n30)
- ***Relays*:** 10 (n31 a n40)
- ***Relays* conectados por nodo:** 4 lógicos, 4 físicos (con latencia de 20 ms)
- ***Relays* conectados por *relay*:** 4 lógicos, 4 físicos (con latencia de 30 ms)
- **Stake:** heterogéneo (siendo distribuido según la figura 4.20)
- **Promedio de *stake* por *relay*:** 10 % (*RSD* 20 %)
- **Tiempo de ejecución:** 10 h
- **Servidor:** local (1 servidor)

Luego de correr el experimento, realizamos el análisis del mismo utilizando la metodología anterior y obtenemos lo siguiente.

Partiendo de los valores de la tabla 4.11, vemos que si bien la precisión es muy alta, el *recall* es muy baja: es decir, estamos reconociendo muy pocas de las cuentas conectadas, pero las que detectamos las predecimos correctamente. Por ejemplo, representamos en la figura 4.21 un caso del mejor resultado (350 ventanas de un minuto, donde el *recall* es del 36 %).

En este caso, de las 12 cuentas conectadas al *relay*, solo reconocemos la mitad, quedando el resto con una predicción errónea. Esto puede deberse a que las seis cuentas contienen un *stake* mayor que las demás, haciendo que la diferencia contra ellas sea más notoria y terminen opacando su colaboración. Nos gustaría mejorar esta estrategia de manera que podamos detectar estas cuentas con menor *stake* y así mejorar el *recall* de nuestro análisis.

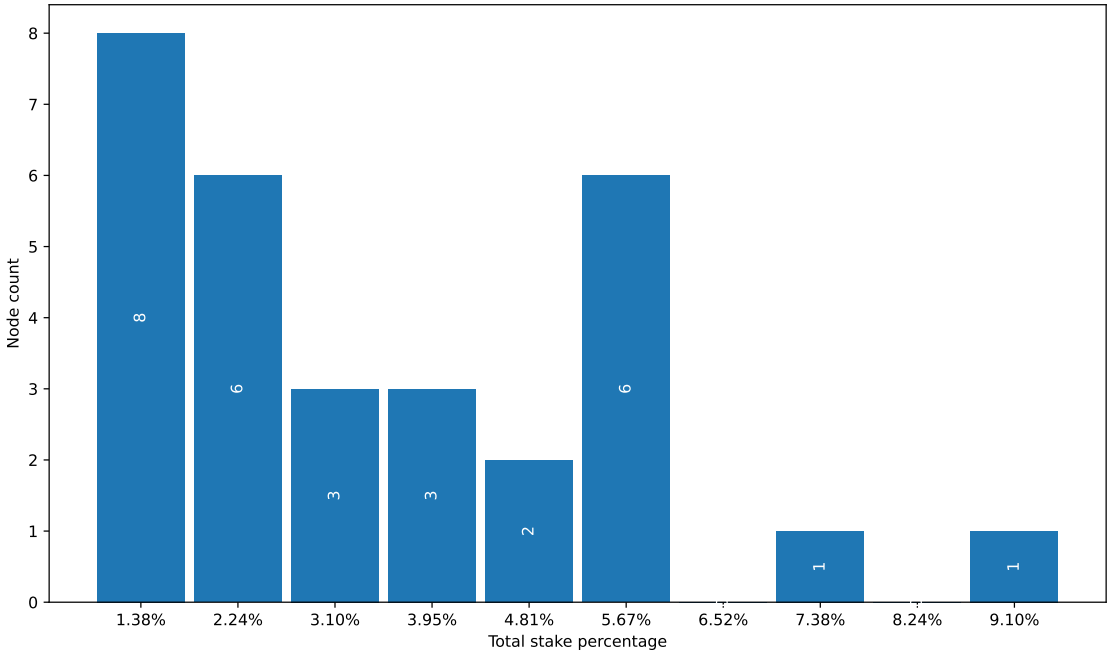


Fig. 4.20: Distribución del *stake* en el sistema según porcentaje del mismo, para 30 nodos y 10 *relays*. Se marca el valor medio de cada intervalo, con un ancho de bin del 0,81 %.

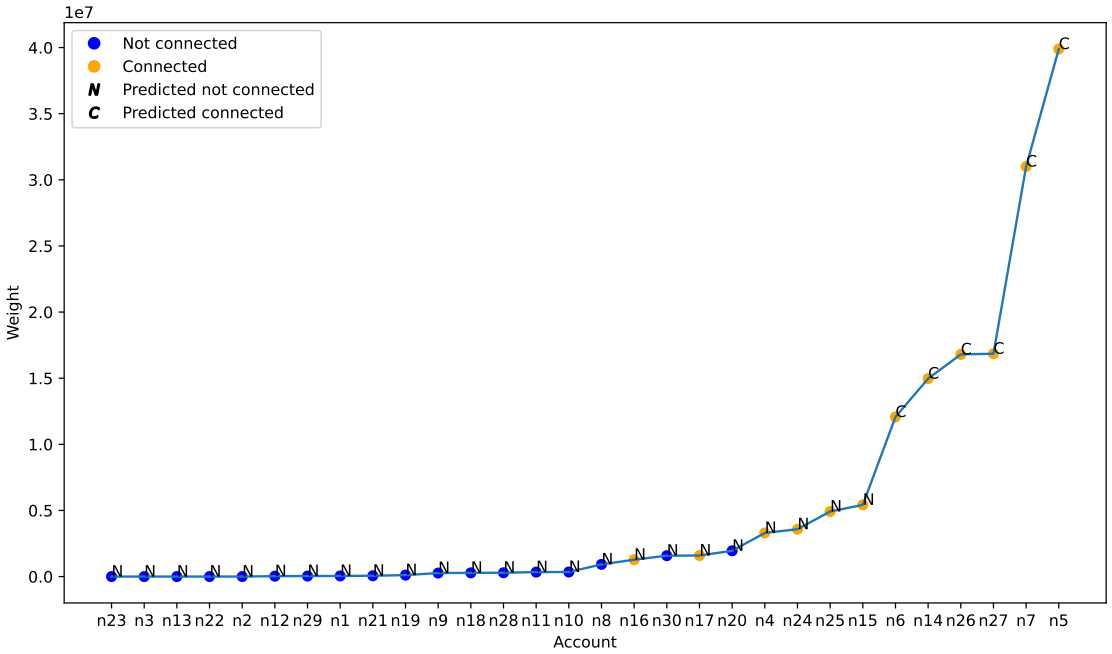


Fig. 4.21: Peso por cuenta para el *relay* n37 tomando 350 ventanas de un minuto cada una, marcados con color azul aquellos no conectados, con naranja los conectados, con una N los predichos como no conectados y una C los predichos como conectados

window size (mins)	window count	total mins	connected precision	connected *recall*
1	5	5	1	0.2037
1	10	10	1	0.2407
1	25	25	1	0.2777
1	50	50	1	0.3055
1	100	100	1	0.2129
1	150	150	1	0.3148
1	200	200	1	0.2870
1	250	250	1	0.3240
1	300	300	1	0.3333
1	350	350	1	0.3611
2	5	10	1	0.1458
2	10	20	1	0.2407
2	25	50	1	0.1851
2	50	100	1	0.25
2	100	200	1	0.2777
2	150	300	1	0.2314
5	5	25	1	0.1944
5	10	50	1	0.2685
5	25	125	1	0.1666
5	50	250	1	0.1666
10	5	50	1	0.2314
10	10	100	1	0.2037
10	25	250	1	0.1851
20	5	100	1	0.2129
20	10	200	1	0.1574
30	5	150	1	0.2129
30	10	300	1	0.1851

Tab. 4.11: Precisión y *recall* según cantidad y tamaño de ventana para la predicción de cuentas conectadas a los *relays* asociados al *relay* n36, para el sistema de 30 nodos y 10 *relays* con *stake* heterogéneo.

4.7.4. Mejorando el peso de los nodos

El problema que vemos es que tenemos una muy mala detección de los nodos que se encuentran en la frontera con los no conectados, que coinciden en su mayoría con los nodos de menor cantidad de *stake* que están conectados al *relay*. Esto se debe a que la cantidad de mensajes que recibimos de ellos termina siendo mucho menor a los de mayor *stake*, y similar a los de gran *stake* que no están conectados.

Luego, queremos de alguna manera ponderar los pesos que les damos a los nodos en cada ventana según su *stake* y analizar si con eso mejora nuestro algoritmo de predicción de conectados. Para esto proponemos dos maneras de realizarlo:

- Ponderar la cantidad de mensajes antes de aplicar los pesos (*count*): con este acercamiento, dentro de cada ventana primero dividimos la cantidad de mensajes recibidos de cada nodo por el *stake* de cada uno de ellos, luego los ordenamos en orden ascendente y finalmente agregamos los pesos de cada nodo según la sucesión de Fibonacci.

- Ponderar los pesos aplicados (*fibo*): a diferencia del anterior, en este caso primero ordenamos por cantidad de mensajes ascendentes los nodos, agregamos los pesos por sucesión de Fibonacci, ponderamos por la cantidad de *stake* del nodo y nuevamente trabajamos con el listado ordenado de manera ascendente.

En ambas estrategias trabajamos aplicando una ponderación por el *stake* de cada nodo, en uno primero ponderando los mensajes recibidos, en otro ponderando los pesos aplicados.

4.7.5. Caso pequeño con nueva estrategia

Analizamos con ambas estrategias el caso planteado anteriormente, obteniendo los resultados presentados en la tabla 4.12 (abreviamos algunas columnas para facilitar la legibilidad, siendo *ws* el *window size*, *prc* la precisión y *rec* el *recall*).

ws(m)	window count	total mins	prc base	rec base	prc count	rec count	prc fibo	rec fibo
1	5	5	1	0.2037	1	0.2037	1	0.4259
1	10	10	1	0.2407	1	0.2222	1	0.5741
1	25	25	1	0.2778	1	0.2593	1	0.6574
1	50	50	1	0.3056	1	0.2593	1	0.8981
1	100	100	1	0.213	1	0.2593	0.9915	0.9444
1	150	150	1	0.3148	1	0.2593	1	0.9722
1	200	200	1	0.287	1	0.2593	1	0.9815
1	250	250	1	0.3241	1	0.2593	0.9915	0.9074
1	300	300	1	0.3333	1	0.2593	0.9915	0.9074
1	350	350	1	0.3611	1	0.2593	0.9915	0.9074
2	5	10	1	0.1458	1	0.2083	1	0.3438
2	10	20	1	0.2407	1	0.213	1	0.4537
2	25	50	1	0.1852	1	0.2593	1	0.7593
2	50	100	1	0.25	1	0.2593	0.9915	0.7593
2	100	200	1	0.2778	1	0.2593	1	0.8241
2	150	300	1	0.2315	1	0.2593	1	0.8241
5	5	25	1	0.1944	1	0.2407	1	0.4722
5	10	50	1	0.2685	1	0.2593	1	0.5556
5	25	125	1	0.1667	1	0.2593	1	0.4815
5	50	250	1	0.1667	1	0.2593	1	0.5926
10	5	50	1	0.2315	1	0.2315	1	0.287
10	10	100	1	0.2037	1	0.2593	1	0.4815
10	25	250	1	0.1852	1	0.2593	1	0.5
20	5	100	1	0.213	1	0.2222	1	0.25
20	10	200	1	0.1574	1	0.213	1	0.4537
30	5	150	1	0.213	1	0.1852	1	0.4167
30	10	300	1	0.1852	1	0.1944	1	0.2778

Tab. 4.12: Precisión y *recall* por estrategia según cantidad y tamaño de ventana para la predicción de cuentas conectadas a los *relays* asociados al *relay* n36, para el sistema de 30 nodos y 10 *relays* con *stake* heterogéneo.

A modo de comparación, vemos el resultado anterior en la columna base. Para el caso de la estrategia *count*, se mantiene la misma precisión que la anterior, pero vemos

que en la mayoría de los casos el *recall* resulta similar o peor que en la estrategia sin ponderado. Tomando el mismo caso analizado para la estrategia base, vemos cómo empeora el resultado en la figura 4.22.

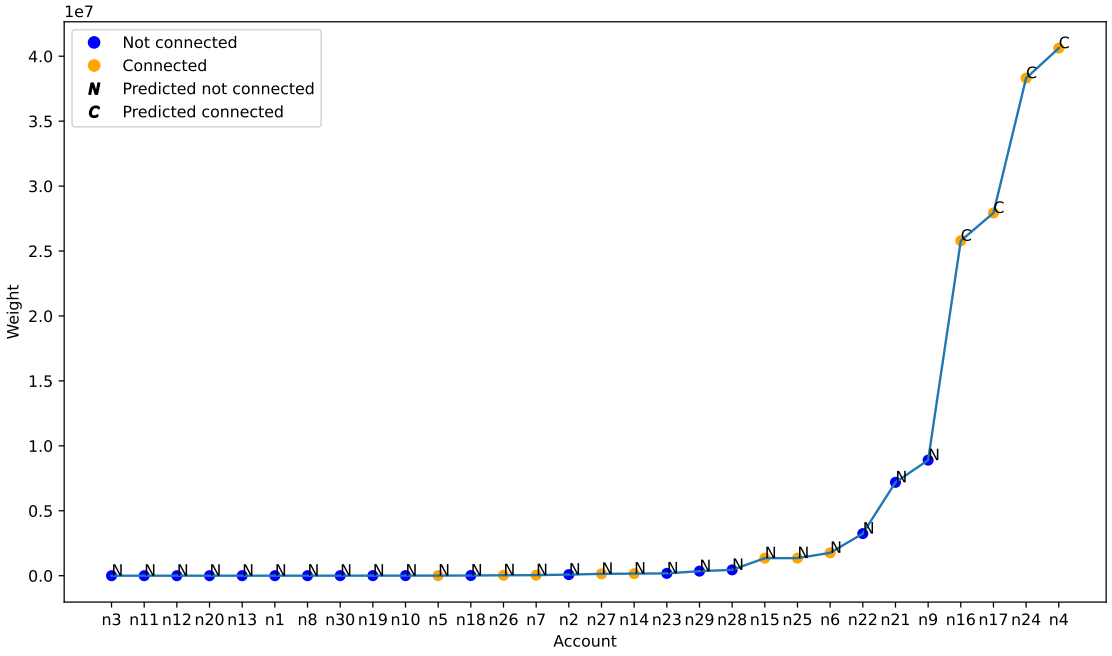


Fig. 4.22: Peso por cuenta para el *relay* n37 tomando 350 ventanas de un minuto cada una, usando estrategia count, marcados con color azul aquellos no conectados, con naranja los conectados, con una N los predichos como no conectados y una C los predichos como conectados.

Esta estrategia no solo disminuye la cantidad de nodos conectados detectados, sino que también mezcla más nodos no conectados hacia la derecha del gráfico, complicando la detección de estos nodos.

En el caso de la estrategia *fibo*, vemos una mejora sustancial en el *recall* para la gran mayoría de los casos, llegando hasta el 98 % tomando 200 ventanas de un minuto. Esto implica que estamos encontrando casi al total de los nodos conectados con una precisión del 100 %, lo cual es un resultado muy prometedor. Los mejores casos se dan para ventanas pequeñas de uno o dos minutos, tomando una gran cantidad de ellas, aunque para las de un minuto no vemos una mejora a partir de las 200.

Para el caso de la ventana analizada previamente, la estrategia logra justamente lo que queríamos según la figura 4.23.

Todos los nodos conectados al *relay* pasan a estar distinguidos a la derecha del gráfico, quedando notoriamente distanciados de los no conectados.

Con este resultado, vemos que la estrategia *count* no logra lo que queremos, mientras que la fibo parece prometedora. Vemos también que obtenemos un mejor resultado tomando muchas ventanas pequeñas en vez de ventanas más grandes (con una diferencia de *recall* muy notoria para el mismo esfuerzo).

Repetimos el experimento, pero con los dos escenarios de *stake* cercano y lejano para comparar lo obtenido en el experimento previo.

Caso pequeño: *stake* cercano

En el caso de *stake* cercano, distribuímos el *stake* conectado a cada *relay* de manera que aquellos conectados directamente al *relay* a analizar manejen las cuentas con mayor participación. Luego, con cada salto, asignamos los nodos de manera que vaya decrementando el *stake* de cada uno. Así, la mayoría del *stake* se encuentra muy cercano al *relay* analizado.

Sistema:

- **Nodos:** 30 (n1 a n30)
- ***Relays*:** 10 (n31 a n40)
- ***Relays* conectados por nodo:** 4 lógicos, 4 físicos (con latencia de 20 ms)
- ***Relays* conectados por *relay*:** 4 lógicos, 4 físicos (con latencia de 30 ms)
- **Stake:** heterogéneo (siendo distribuido según la figura 4.24)
- **Distribución de *stake* por *relay*:** decremental desde el *relay* analizado (la mayoría del *stake* se encuentra **cercano** al *relay*).
- **Tiempo de ejecución:** 10 h
- **Servidor:** local (1 servidor)

El resultado de este experimento, planteado en la tabla 4.13, nos demuestra nuevamente la efectividad del algoritmo fibo en comparación con el base y count: logramos precisión y *recall* perfectos para muestras de pocas ventanas pequeñas. Tomando diez ventanas de un minuto, el algoritmo fibo ya logra el 100 % de precisión y *recall*.

El algoritmo count no mejora con el incremento del tamaño ni la cantidad de ventanas tomadas, otorgando peores resultados que la estrategia base. Por otro lado, la estrategia fibo obtiene mejores resultados con ventanas pequeñas que con más grandes (al igual que

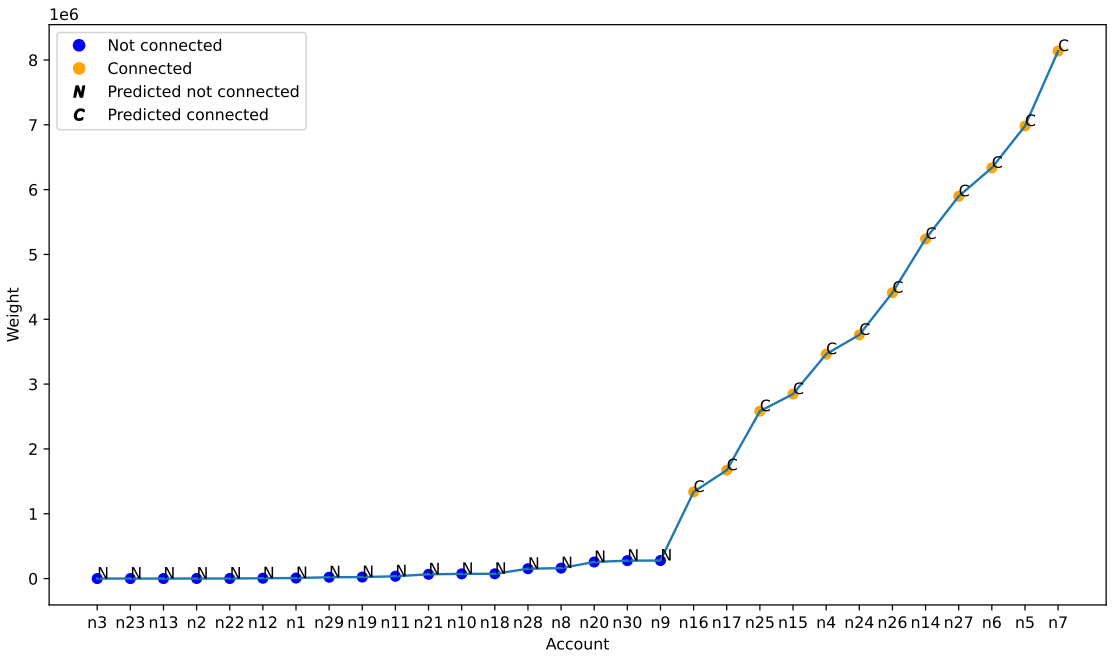


Fig. 4.23: Peso por cuenta para el *relay* n37 tomando 350 ventanas de un minuto cada una, usando estrategia fibo, marcados con color azul aquellos no conectados, con naranja los conectados, con una N los predichos como no conectados y una C los predichos como conectados

en el sistema anterior).

Caso pequeño: *stake* lejano

Vemos entonces qué sucede en un sistema donde el *stake* se incorpora de manera contraria a este caso, asignando las cuentas con mayor *stake* lo más alejadas posible del *relay* analizado.

Sistema:

- **Nodos:** 30 (n1 a n30)
- ***Relays*:** 10 (n31 a n40)
- ***Relays* conectados por nodo:** 4 lógicos, 4 físicos (con latencia de 20 ms)
- ***Relays* conectados por *relay*:** 4 lógicos, 4 físicos (con latencia de 30 ms)
- **Stake:** heterogéneo (siendo distribuido según la figura 4.25)
- **Distribución de *stake* por *relay*:** incremental desde el *relay* analizado (la mayoría del *stake* se encuentra **lejano** al *relay*).
- **Tiempo de ejecución:** 10 h
- **Servidor:** local (1 servidor)

En este sistema con el *stake* lejano, siguiendo los resultados de la tabla 4.14, notamos un fuerte decremento en las métricas de la estrategia base: lo que antes era una precisión y *recall* perfectos, ahora disminuye notablemente. La estrategia count logra mejorar el *recall* de muchos de los casos (inclusive comparado con el escenario cercano), pero solo alcanza el 27 % con una precisión del mismo valor.

La estrategia fibo en este caso nuevamente resulta la mejor, logrando una precisión del 72 % y *recall* del 85 % tomando diez ventanas de un minuto. Tanto con esta estrategia,

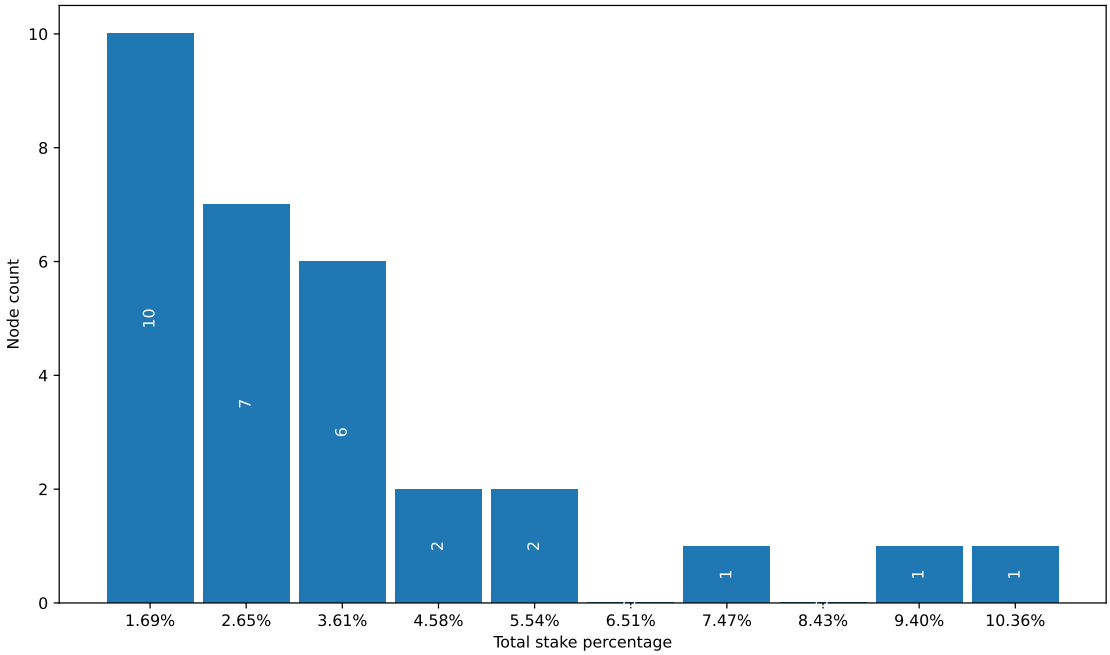


Fig. 4.24: Distribución del *stake* en el sistema según porcentaje del mismo, para 30 nodos y 10 *relays*, con mayoría de *stake* **cercano**. Se marca el valor medio de cada intervalo, con un ancho de bin del 0,91 %.

ws(m)	window count	total mins	prc base	rec base	prc count	rec count	prc fibo	rec fibo
1	5	5	1	0.3333	0.25	0.0833	1	0.9167
1	10	10	1	0.4167	0.25	0.0833	1	1
1	25	25	1	1	0.25	0.0833	1	1
1	50	50	1	1	0.25	0.0833	1	1
1	100	100	1	1	0.25	0.0833	1	1
1	150	150	1	1	0.25	0.0833	1	1
1	200	200	1	0.8958	0.25	0.0833	1	1
1	250	250	1	1	0.25	0.0833	1	1
1	300	300	1	1	0.25	0.0833	1	1
1	350	350	1	1	0.25	0.0833	1	1
1	400	400	1	1	0.25	0.0833	1	1
1	450	450	1	1	0.25	0.0833	1	1
1	500	500	1	1	0.25	0.0833	1	1
2	5	10	1	0.25	0.25	0.0833	1	0.7708
2	10	20	1	0.4792	0.25	0.0833	1	0.9167
2	25	50	1	0.3542	0.25	0.0833	1	1
2	50	100	1	0.3125	0.25	0.0833	1	1
2	100	200	1	0.3125	0.25	0.0833	1	1
2	150	300	1	0.3542	0.25	0.0833	1	1
2	200	400	1	0.375	0.25	0.0833	1	1
2	250	500	1	0.375	0.25	0.0833	1	1
5	5	25	1	0.2292	0.25	0.0833	1	0.3333
5	10	50	1	0.1667	0.25	0.0833	1	0.5208
5	25	125	1	0.2708	0.25	0.0833	1	0.6042
5	50	250	1	0.2917	0.25	0.0833	1	0.6042
5	100	500	1	0.2917	0.25	0.0833	1	0.8125
10	5	50	1	0.25	0.25	0.0833	1	0.3125
10	10	100	1	0.2708	0.25	0.0833	1	0.6042
10	25	250	1	0.2708	0.25	0.0833	1	0.5208
10	50	500	1	0.2708	0.25	0.0833	1	0.5208
20	5	100	1	0.0833	0.25	0.0833	1	0.3125
20	10	200	1	0.2083	0.25	0.0833	1	0.3125
20	25	500	1	0.2292	0.25	0.0833	1	0.5208
30	5	150	1	0.1458	0.25	0.0833	1	0.2292
30	10	300	1	0.2292	0.25	0.0833	1	0.3125

Tab. 4.13: Precisión y *recall* por estrategia según cantidad y tamaño de ventana para la predicción de cuentas conectadas a los *relays* asociados al *relay* n31, para el sistema de 30 nodos y 10 *relays* con *stake* heterogéneo **cercano**.

como con count, no notamos una mejora al incrementar tanto el tamaño como la cantidad de ventanas tomadas.

A pesar de que la estrategia fibo continúa siendo la mejor opción, notamos un decremento en las métricas en comparación con el caso de *stake* cercano. El hecho de alejar el *stake* genera una complejidad adicional para poder predecir la ubicación de los nodos: en el caso cercano, los nodos que emiten la mayor cantidad de mensajes del sistema se en-

cuentran conectados a *relays* que se comunican directamente con el *relay* analizado. Así, estos mensajes llegan inmediatamente al *relay* sin pasar por intermediarios. En el caso lejano, se agregan *relays* en el camino y eso complica el análisis de la información.

A medida que incrementemos el tamaño de los sistemas querremos verificar si se mantiene esta tendencia.

4.7.6. Caso mediano

Como primer paso para escalar esta experiencia, repetimos la escala de la distribución homogénea y corremos un experimento con 60 nodos y 20 *relays*, el doble de la red previa.

Sistema:

- **Nodos:** 60 (n1 a n60)
- ***Relays*:** 20 (n61 a n80)
- ***Relays* conectados por nodo:** 4 lógicos, 4 físicos (con latencia de 20 ms)
- ***Relays* conectados por *relay*:** 4 lógicos, 8 físicos (con latencia de 30 ms)
- **Stake:** heterogéneo (siendo distribuido según la figura 4.26)
- **Promedio de *stake* por *relay*:** 5 % (*RSD* 20 %)
- **Tiempo de ejecución:** 10 h
- **Servidor:** local (1 servidor)

Con esta configuración obtenemos los resultados expresados en la tabla 4.15.

Con este tamaño de experimento vemos que la precisión base resulta muy buena, pero el *recall* solo llega al 50 % en algunos casos. Con la estrategia count no vemos una mejora en las métricas, de hecho resultan peores en la mayoría de los casos. Mientras tanto, con la estrategia fibo vemos que a partir de una cierta cantidad de ventanas tomadas, el

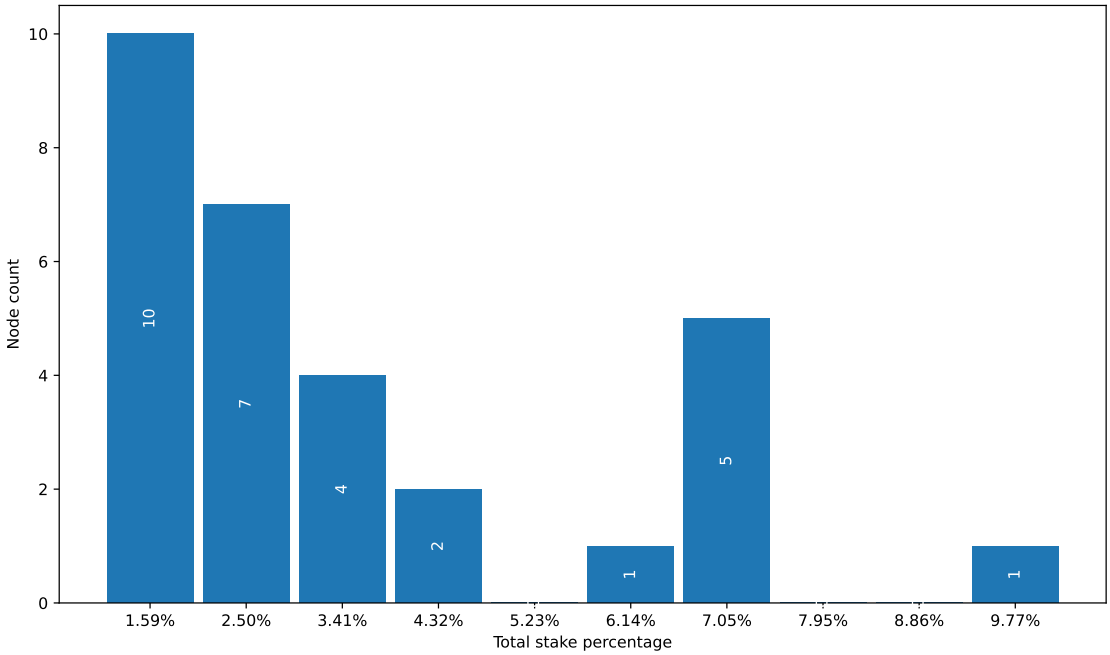


Fig. 4.25: Distribución del *stake* en el sistema según porcentaje del mismo, para 30 nodos y 10 *relays*, con mayoría de *stake* **lejano**. Se marca el valor medio de cada intervalo, con un ancho de bin del 0,86 %.

ws(m)	window count	total mins	prc base	rec base	prc count	rec count	prc fibo	rec fibo
1	5	5	0.25	0.1875	0.25	0.2292	0.5625	0.7907
1	10	10	0.25	0.0625	0.25	0.1875	0.7292	0.855
1	25	25	0.25	0.0208	0.2308	0.25	0.7292	0.855
1	50	50	0.25	0.0208	0.275	0.2708	0.7292	0.855
1	100	100	0.25	0.0625	0.25	0.25	0.7292	0.855
1	150	150	0.25	0.0625	0.25	0.25	0.7292	0.855
1	200	200	0.25	0.1458	0.25	0.25	0.7292	0.855
1	250	250	0.25	0.0625	0.25	0.25	0.7292	0.855
1	300	300	0.25	0.0625	0.25	0.25	0.7292	0.855
1	350	350	0.25	0.1458	0.25	0.25	0.7292	0.855
1	400	400	0.25	0.1458	0.25	0.25	0.7292	0.855
1	450	450	0.25	0.1458	0.25	0.25	0.7292	0.855
1	500	500	0.25	0.1458	0.25	0.25	0.7292	0.855
2	5	10	0.25	0.1875	0.25	0.1875	0.7292	0.855
2	10	20	0.25	0.0208	0.25	0.1042	0.7292	0.855
2	25	50	0.25	0.0625	0.25	0.0625	0.7292	0.855
2	50	100	0.25	0.0208	0.25	0.1458	0.7292	0.855
2	100	200	0.25	0.0625	0.25	0.25	0.7292	0.855
2	150	300	0.25	0.0625	0.25	0.25	0.7292	0.855
2	200	400	0.25	0.0625	0.25	0.25	0.7292	0.855
2	250	500	0.25	0.1458	0.25	0.25	0.7292	0.855
5	5	25	0.25	0.0625	0.25	0.1667	0.7292	0.855
5	10	50	0.25	0.0417	0.25	0.0417	0.7292	0.855
5	25	125	0.25	0.1458	0.25	0.1458	0.7292	0.855
5	50	250	0.25	0.1458	0.25	0.1458	0.7292	0.855
5	100	500	0.25	0.0625	0.25	0.1458	0.7292	0.855
10	5	50	0.25	0.0625	0.25	0.1667	0.5	0.752
10	10	100	0.25	0.0208	0.25	0.0208	0.7292	0.855
10	25	250	0.25	0.0625	0.25	0.1458	0.7292	0.855
10	50	500	0.25	0.0625	0.25	0.0625	0.7292	0.855
20	5	100	0.25	0.0208	0.25	0.0833	0.7292	0.855
20	10	200	0.25	0.0625	0.25	0.1458	0.7083	0.8443
20	25	500	0.25	0.0625	0.25	0.1458	0.7292	0.855
30	5	150	0.25	0.0625	0.25	0.0625	0.6875	0.8336
30	10	300	0.25	0.0625	0.25	0.0625	0.7292	0.855

Tab. 4.14: Precisión y *recall* por estrategia según cantidad y tamaño de ventana para la predicción de cuentas conectadas a los *relays* asociados al *relay* n31, para el sistema de 30 nodos y 10 *relays* con *stake* heterogéneo **lejano**.

recall aumenta considerablemente y logramos muy buenos resultados con esfuerzos de 150 minutos: tomando 150 ventanas de un minuto se logra un 98 % de precisión y un 91 % de *recall*. Con ventanas de dos minutos logramos resultados similares, obteniendo buenas métricas con esfuerzos equivalentes. Pero para ventanas de cinco minutos y mayores, el *recall* no mejora como veníamos observando y el esfuerzo resulta muy superior al necesario para las otras ventanas.

La estrategia de fibo nos otorga mejores resultados en ambos experimentos, con mejoras notables al poder diferenciar mejor los nodos conectados de los no conectados a los *relays* analizados, incrementando esa diferencia y favoreciendo nuestro algoritmo de reconocimiento.

Caso mediano: *stake* cercano

Experimentamos entonces con una red donde el *stake* se encuentra cercano al *relay* analizado.

Sistema:

- **Nodos:** 60 (n1 a n60)
- ***Relays*:** 20 (n61 a n80)
- ***Relays* conectados por nodo:** 4 lógicos, 4 físicos (con latencia de 20 ms)
- ***Relays* conectados por *relay*:** 4 lógicos, 8 físicos (con latencia de 30 ms)
- **Stake:** heterogéneo (siendo distribuido según la figura 4.27)
- **Distribución de *stake* por *relay*:** decremental desde el *relay* analizado (la mayoría del *stake* se encuentra **cercano** al *relay*).
- **Tiempo de ejecución:** 10 h
- **Servidor:** local (1 servidor)

Para este sistema con *stake* cercano, vemos en la tabla 4.16 que obtenemos mejores resultados para las estrategias base y fibo que en el caso anterior. Aquí ya con 25 ventanas de un minuto logramos excelentes resultados tanto en precisión como en *recall* para ambas estrategias (más del 91 % de precisión y 97 % de *recall*). Tomando ventanas pequeñas el algoritmo fibo no mejora los resultados, pero se mantiene muy cercano a lo obtenido por la base. Para ventanas más grandes, a partir de los cinco minutos, notamos una mejora en

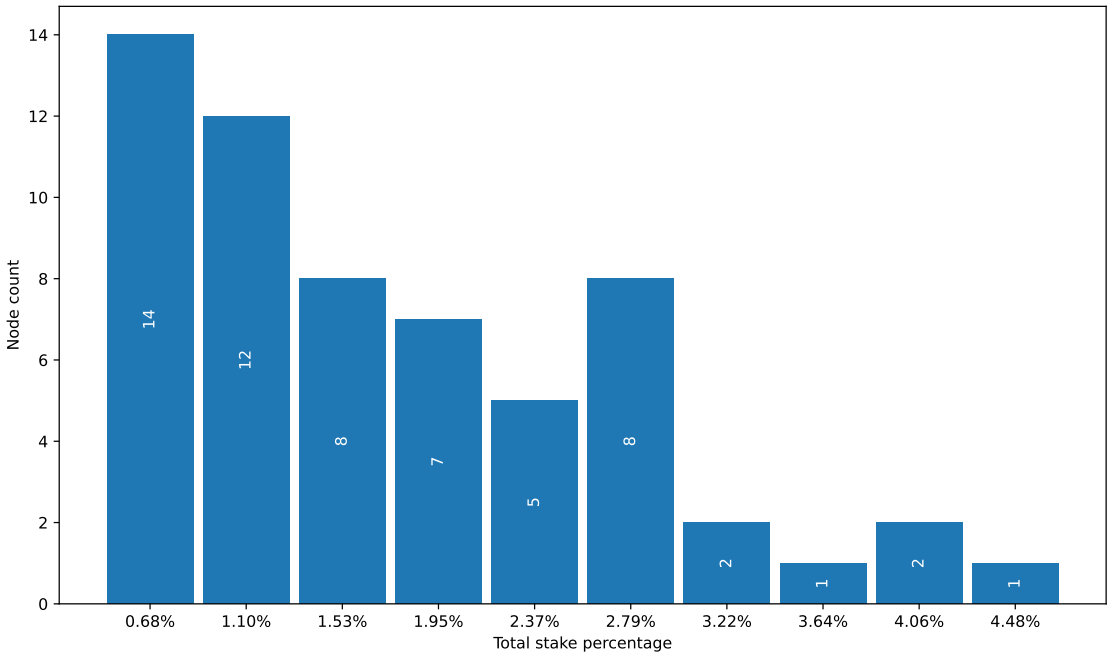


Fig. 4.26: Distribución del *stake* en el sistema según porcentaje del mismo, para 60 nodos y 20 *relays*. Se marca el valor medio de cada intervalo, con un ancho de bin del 0,37 %.

ws(m)	window count	total mins	prc base	rec base	prc count	rec count	prc fibo	rec fibo
1	5	5	0.9841	0.2262	0.7946	0.1845	0.9513	0.4821
1	10	10	0.9568	0.3158	0.8289	0.1579	0.9311	0.4386
1	25	25	0.9808	0.3728	0.8955	0.2105	0.9381	0.7018
1	50	50	0.9645	0.4386	0.9192	0.2149	0.957	0.8026
1	100	100	0.9904	0.443	0.8672	0.2368	0.996	0.8772
1	150	150	1	0.4561	0.9386	0.2325	0.9859	0.9123
1	200	200	0.9785	0.5	0.8885	0.2325	0.9806	0.9167
1	250	250	0.9785	0.5132	0.9323	0.2368	0.9919	0.9298
1	300	300	0.9889	0.4781	0.9323	0.2368	0.9831	0.9693
1	350	350	0.9947	0.4693	0.8935	0.2368	0.9912	0.9211
2	5	10	1	0.2451	0.9412	0.1618	0.971	0.4461
2	10	20	0.9753	0.2731	0.9	0.1667	0.9748	0.5602
2	25	50	0.9759	0.4254	0.8073	0.2368	0.9722	0.7719
2	50	100	0.9852	0.4123	0.8838	0.2368	0.9881	0.8596
2	100	200	0.9947	0.3947	0.9035	0.2368	0.9912	0.9254
2	150	300	0.9947	0.3772	0.9035	0.2368	0.9793	0.9737
5	5	25	0.9869	0.2549	0.9265	0.2059	0.9769	0.3873
5	10	50	0.9895	0.2456	0.885	0.2193	0.9323	0.4649
5	25	125	1	0.2719	0.9474	0.2368	0.9811	0.6711
5	50	250	1	0.307	0.9386	0.2368	0.9831	0.864
10	5	50	0.9907	0.2407	0.9083	0.2176	0.9828	0.3796
10	10	100	1	0.2149	0.9342	0.2368	1	0.6316
10	25	250	1	0.2368	0.9211	0.2368	0.9853	0.5965
20	5	100	0.9889	0.2222	0.9028	0.2222	0.9921	0.4444
20	10	200	1	0.2281	0.9211	0.2281	1	0.5263
30	5	150	1	0.2149	0.9474	0.2281	1	0.3246
30	10	300	1	0.2544	0.9474	0.2368	0.996	0.6053

Tab. 4.15: Precisión y *recall* por estrategia según cantidad y tamaño de ventana para la predicción de cuentas conectadas a los *relays* asociados al *relay* n62, para el sistema de 60 nodos y 20 *relays* con *stake* heterogéneo.

el *recall* en fibo, pero a medida que se agrandan las ventanas, el valor del *recall* cae para ambas.

La estrategia count falla absolutamente, sin lograr predecir ni un solo caso a pesar de tomar muestras de distintos tamaños y cantidades.

Caso pequeño: *stake* lejano

Repetimos la experiencia, pero esta vez asignando el *stake* lo más alejado posible del *relay* analizado.

Sistema:

- **Nodos:** 60 (n1 a n60)
- ***Relays*:** 20 (n61 a n80)
- ***Relays* conectados por nodo:** 4 lógicos, 4 físicos (con latencia de 20 ms)
- ***Relays* conectados por *relay*:** 4 lógicos, 8 físicos (con latencia de 30 ms)
- **Stake:** heterogéneo (siendo distribuido según la figura 4.28)

- **Distribución de *stake* por *relay*:** incremental desde el *relay* analizado (la mayoría del *stake* se encuentra **lejano** al *relay*).
- **Tiempo de ejecución:** 10 h
- **Servidor:** local (1 servidor)

Para esta experiencia con *stake* lejano, los resultados reflejados en la tabla 4.17 nos demuestran que el algoritmo base presenta dificultad para la detección de conectados en comparación con el caso cercano. Sin embargo, la estrategia fibo mantiene muy buenos resultados nuevamente con poca información: con 25 ventanas de un minuto ya logramos una precisión del 89 %, y un *recall* del 93 %. Luego, estos resultados no mejoran al incrementar la muestra con ventanas pequeñas.

En el caso de la estrategia count, vemos que el resultado mejora en relación al escenario cercano, donde la predicción fue nula. Aquí se lograron predicciones, pero con valores que continúan siendo malos, sobre todo en comparación con la estrategia fibo.

Al igual que en los escenarios pequeños, la estrategia fibo resulta la óptima, con valores muy prometedores para ambas métricas. A diferencia de lo que sucedía en el tamaño anterior, no vemos que la predicción con esta estrategia se vea sumamente afectada por la ubicación del *stake*.

4.7.7. Escalando el experimento: caso semi-realista

Al igual que con el escenario de *stake* homogéneo, realizamos un experimento con 150 nodos y 50 *relays* con *stake* heterogéneo para evaluar la eficacia de nuestras predicciones.

Sistema:

- **Nodos:** 150 (n1 a n150)

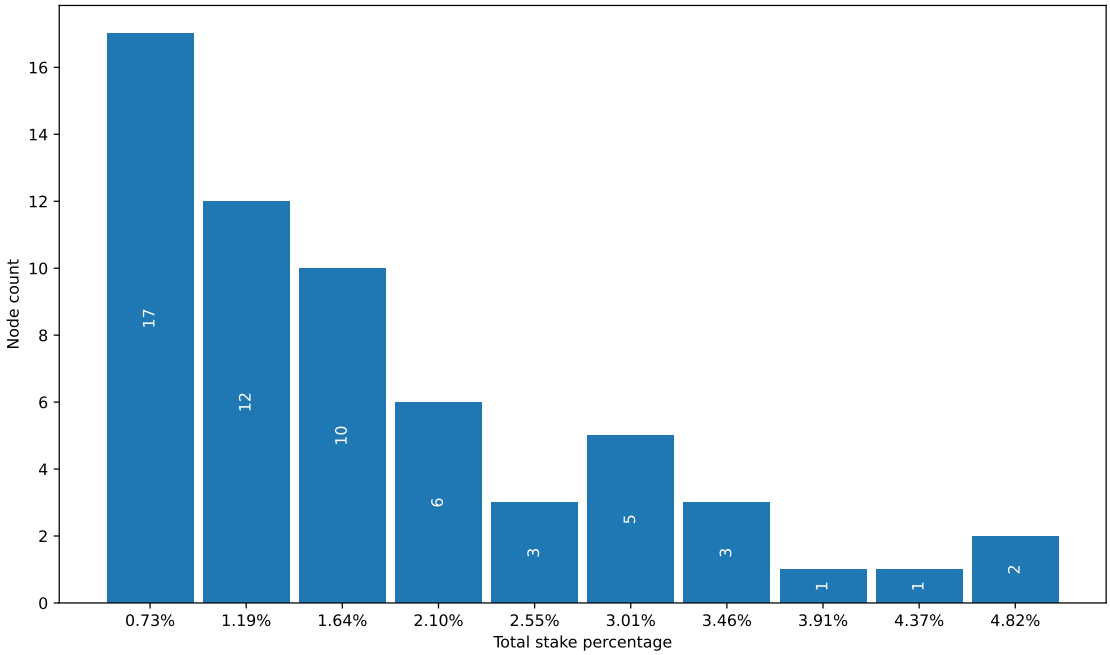


Fig. 4.27: Distribución del *stake* en el sistema según porcentaje del mismo, para 60 nodos y 20 *relays*, con mayoría de *stake* **cercano**. Se marca el valor medio de cada intervalo, con un ancho de bin del 0,40 %.

ws(m)	window count	total mins	prc base	rec base	prc count	rec count	prc fibo	rec fibo
1	5	5	0.925	0.5833	0	0	0.9167	0.5625
1	10	10	0.9286	0.8542	0	0	0.9063	0.8125
1	25	25	0.9464	0.9792	0	0	0.9118	0.9792
1	50	50	0.95	1	0	0	0.95	1
1	100	100	1	1	0	0	0.9079	1
1	150	150	0.9643	1	0	0	0.9	1
1	200	200	0.9808	1	0	0	0.9	1
1	250	250	0.9808	1	0	0	0.9	1
1	300	300	0.9808	1	0	0	0.9167	1
1	350	350	0.9808	1	0	0	0.9167	1
1	400	400	0.9808	1	0	0	0.9167	1
1	450	450	0.9808	1	0	0	0.9167	1
1	500	500	0.9808	1	0	0	0.9167	1
2	5	10	1	0.3542	0	0	1	0.3333
2	10	20	0.9643	0.9792	0	0	0.975	0.9167
2	25	50	0.9808	1	0	0	0.9643	1
2	50	100	1	1	0	0	1	1
2	100	200	0.9808	1	0	0	1	1
2	150	300	0.9808	1	0	0	0.9375	1
2	200	400	0.9808	1	0	0	0.9375	1
2	250	500	0.9808	1	0	0	0.9375	1
5	5	25	1	0.5625	0	0	1	0.6667
5	10	50	1	0.6042	0	0	1	0.8125
5	25	125	1	0.5208	0	0	1	1
5	50	250	1	0.6042	0	0	1	1
5	100	500	1	0.8542	0	0	1	1
10	5	50	1	0.2708	0	0	1	0.3333
10	10	100	1	0.5417	0	0	1	0.5417
10	25	250	1	0.3542	0	0	1	0.5625
10	50	500	1	0.2917	0	0	1	0.6042
20	5	100	1	0.25	0	0	1	0.2708
20	10	200	1	0.3542	0	0	1	0.3542
20	25	500	1	0.2708	0	0	1	0.2083
30	5	150	1	0.0833	0	0	1	0.1875
30	10	300	1	0.25	0	0	1	0.2917

Tab. 4.16: Precisión y *recall* por estrategia según cantidad y tamaño de ventana para la predicción de cuentas conectadas a los *relays* asociados al *relay* n61, para el sistema de 60 nodos y 20 *relays* con *stake* heterogéneo **cercano**.

- ***Relays*:** 50 (n151 a n200)
- ***Relays* conectados por nodo:** 4 lógicos, 4 físicos (con latencia de 20 ms)
- ***Relays* conectados por *relay*:** 4 lógicos, 18 físicos (con latencia de 30 ms)
- **Stake:** heterogéneo (siendo distribuido según la figura 4.29)
- **Promedio de *stake* por *relay*:** 2 % (*RSD* 20 %)
- **Tiempo de ejecución:** 10 h

- **Servidor:** CloudLab (5 servidores *d6515*)

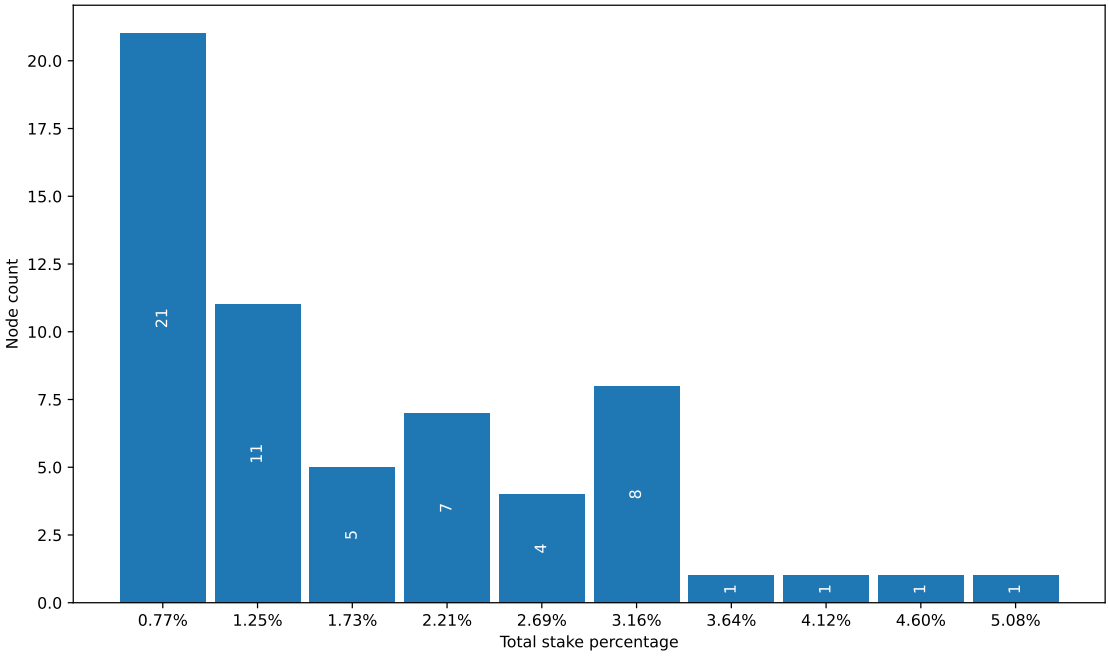


Fig. 4.28: Distribución del *stake* en el sistema según porcentaje del mismo, para 60 nodos y 20 *relays*, con mayoría de *stake* **lejano**. Se marca el valor medio de cada intervalo, con un ancho de bin del 0,43 %.

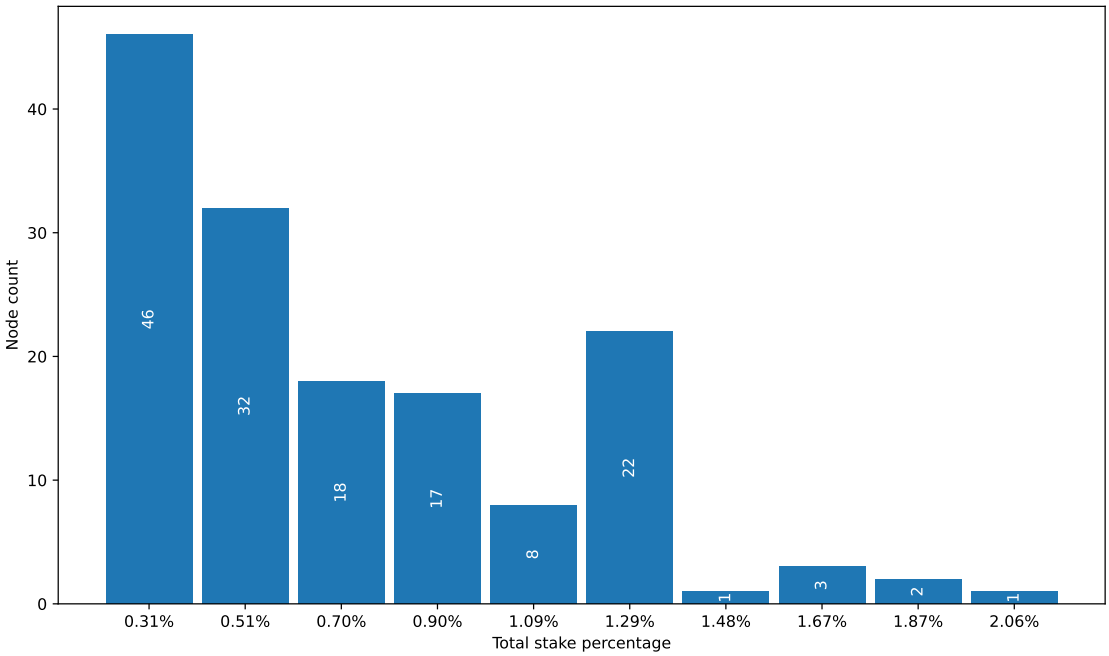


Fig. 4.29: Distribución del *stake* en el sistema según porcentaje del mismo, para 150 nodos y 50 *relays*. Se marca el valor medio de cada intervalo, con un ancho de *bin* del 0,15 %.

ws(m)	window count	total mins	prc base	rec base	prc count	rec count	prc fibo	rec fibo
1	5	5	0.4583	0.0833	0.75	0.0625	1	0.1875
1	10	10	0.159	0.25	0.6667	0.1458	0.8929	0.7708
1	25	25	0.1611	0.2708	0.4702	0.5	0.8929	0.9375
1	50	50	0.2843	0.5	0.392	0.7083	0.8929	0.9375
1	100	100	0.3891	0.4375	0.6103	0.5625	0.8929	0.9375
1	150	150	0.325	0.4583	0.5784	0.5833	0.8929	0.9375
1	200	200	0.2833	0.3958	0.4048	0.75	0.8929	0.9375
1	250	250	0.2815	0.4167	0.5833	0.5833	0.8929	0.9375
1	300	300	0.2429	0.75	0.4048	0.75	0.8929	0.9375
1	350	350	0.2817	0.6042	0.4048	0.75	0.8929	0.9375
1	400	400	0.2679	0.625	0.4048	0.75	0.8929	0.9375
1	450	450	0.2679	0.625	0.4048	0.75	0.8929	0.9375
1	500	500	0.2902	0.4375	0.4048	0.75	0.8929	0.9375
2	5	10	0.4375	0.1667	0.5909	0.1458	0.925	0.625
2	10	20	0.4182	0.25	0.6346	0.2292	1	0.7083
2	25	50	0.2112	0.25	0.3843	0.5208	0.8929	0.9375
2	50	100	0.2455	0.3125	0.5667	0.5208	0.8929	0.9375
2	100	200	0.1777	0.2917	0.4077	0.7292	0.8929	0.9375
2	150	300	0.2381	0.5	0.4048	0.75	0.8929	0.9375
2	200	400	0.1786	0.3125	0.4048	0.75	0.8929	0.9375
2	250	500	0.1964	0.3542	0.4048	0.75	0.8929	0.9375
5	5	25	0.125	0.0208	0.55	0.3125	0.9643	0.5208
5	10	50	0.3214	0.0833	0.4219	0.2708	0.9028	0.9167
5	25	125	0.1592	0.1667	0.2633	0.25	0.8929	0.9375
5	50	250	0.1615	0.1667	0.4063	0.5833	0.8929	0.9375
5	100	500	0.1071	0.0625	0.4271	0.625	0.8929	0.9375
10	5	50	0.2727	0.0417	0.4327	0.1667	0.9722	0.4375
10	10	100	0.2778	0.0625	0.3	0.1042	0.9423	0.875
10	25	250	0.25	0.0625	0.3125	0.1042	0.9	0.9375
10	50	500	0.25	0.0625	0.4881	0.3958	0.8929	0.9375
20	5	100	0.25	0.0625	0.2917	0.0833	0.975	0.7292
20	10	200	0.25	0.0625	0.25	0.0625	1	0.7083
20	25	500	0.25	0.0625	0.25	0.0625	0.8929	0.9375
30	5	150	0.25	0.0417	0.25	0.0417	0.9375	0.8542
30	10	300	0.25	0.0625	0.25	0.0625	0.9167	0.9375

Tab. 4.17: Precisión y *recall* por estrategia según cantidad y tamaño de ventana para la predicción de cuentas conectadas a los *relays* asociados al *relay* n61, para el sistema de 60 nodos y 20 *relays* con *stake* heterogéneo **lejano**.

Con este tamaño de red, vemos en la tabla 4.18 que los resultados no son tan promisorios: si bien logramos muy buena precisión con la estrategia base y la *fibo*, la segunda mejora el *recall* de la primera en más del doble en algunos casos, pero sin superar el 47 %. De esta manera, observamos que si bien nuestra predicción es muy certera, estamos perdiendo alrededor de la mitad de los nodos efectivamente conectados a cada uno de los *relays* con los que se comunica nuestro atacado. La estrategia *count* demuestra una vez

ws(m)	window count	total mins	prc base	rec base	prc count	rec count	prc fibo	rec fibo
1	5	5	0.8333	0.0833	0.6369	0.2222	0.75	0.3056
1	10	10	0.7381	0.25	0.5556	0.3333	0.7879	0.3056
1	25	25	0.8056	0.2222	0.547	0.4444	1	0.3889
1	50	50	0.8889	0.1667	0.5795	0.4444	1	0.3611
1	100	100	0.8056	0.2222	0.5734	0.4444	1	0.3056
1	150	150	0.8889	0.1667	0.4983	0.4444	0.9444	0.3056
1	200	200	0.8056	0.2222	0.5918	0.4444	1	0.3333
1	250	250	0.8889	0.1667	0.5452	0.4444	0.963	0.4167
1	300	300	0.8889	0.1667	0.323	0.4444	0.963	0.4167
1	350	350	0.8056	0.2222	0.2357	0.3333	0.9583	0.4722
1	400	400	0.8056	0.2222	0.3059	0.4444	0.9583	0.4722
1	450	450	0.8056	0.2222	0.331	0.4444	0.9583	0.4722
1	500	500	0.8056	0.2222	0.2913	0.4444	1	0.3889
1	600	600	0.8056	0.2222	0.4899	0.4444	1	0.2778
2	5	10	1	0.1667	0.4841	0.2778	0.9167	0.1389
2	10	20	1	0.1667	0.6667	0.1667	1	0.3056
2	25	50	1	0.1111	0.3134	0.4444	1	0.4444
2	50	100	0.9167	0.1389	0.3909	0.4444	1	0.3889
2	100	200	0.8889	0.1667	0.2889	0.4444	1	0.3889
2	150	300	0.9167	0.1389	0.2773	0.4444	1	0.3889
2	200	400	0.8889	0.2222	0.3009	0.4444	1	0.3889
2	250	500	0.8889	0.1667	0.7273	0.4444	1	0.3889
5	5	25	0.9167	0.1667	0.8	0.25	0.8611	0.2778
5	10	50	0.7222	0.1667	0.4389	0.3889	0.9167	0.1667
5	25	125	0.9167	0.1944	0.5791	0.4444	1	0.3611
5	50	250	0.9167	0.1944	0.6717	0.4444	1	0.3611
5	100	500	0.9167	0.1944	0.5606	0.4444	1	0.3611
10	5	50	0.8889	0.1667	0.4028	0.3056	1	0.2778
10	10	100	0.9167	0.1389	0.8	0.1667	0.9333	0.3889
10	25	250	0.9167	0.3333	0.5952	0.4444	1	0.2778
10	50	500	0.9167	0.1944	0.9444	0.4444	1	0.3333
20	5	100	1	0.1111	0.637	0.4167	1	0.3056
20	10	200	1	0.0833	0.6852	0.4444	1	0.25
20	25	500	1	0.0833	1	0.4444	1	0.2778
30	5	150	1	0.2222	1	0.2222	1	0.2222
30	10	300	1	0.0833	1	0.4167	0.9333	0.2222

Tab. 4.18: Precisión y *recall* por estrategia según cantidad y tamaño de ventana para la predicción de cuentas conectadas a los *relays* asociados al *relay* n151, para el sistema de 150 nodos y 50 *relays* con *stake* heterogéneo.

más no dar buenos resultados, si bien mejora el *recall*, empeora notablemente la precisión de la predicción (básicamente es más permisiva con los nodos que predice como conectados, aumentando el *pool* de conectados, pero también agregando más falsos positivos).

El mejor resultado lo obtenemos para 350 ventanas de un minuto, requiriendo un esfuerzo de 350, mientras que con 25 ventanas de dos minutos nos acercamos fuertemente

a los mismos valores, pero con un esfuerzo de solo 50.

Se entiende que este resultado demuestra la complejidad de lo que queremos detectar, no es trivial encontrar los nodos conectados al *relay* cuando la probabilidad de los mensajes de cada uno es desigual dependiendo de su *stake*. Si bien a priori conocemos esa distribución, el algoritmo resulta insuficiente para poder hacer una buena predicción en base a la información recogida.

Continuamos los experimentos con los dos escenarios de *stake* cercano y lejano del *relay* analizado.

Caso semi-realista: *stake* cercano

Buscamos analizar qué sucede en un sistema de este tamaño donde el *stake* se encuentra lo más cercano posible al *relay* a analizar.

Sistema:

- **Nodos:** 150 (n1 a n150)
- ***Relays*:** 50 (n151 a n200)
- ***Relays* conectados por nodo:** 4 lógicos, 4 físicos (con latencia de 20 ms)
- ***Relays* conectados por *relay*:** 4 lógicos, 18 físicos (con latencia de 30 ms)
- **Stake:** heterogéneo (siendo distribuido según la figura 4.30)
- **Distribución de *stake* por *relay*:** incremental desde el *relay* analizado (la mayoría del *stake* se encuentra **cercano** al *relay*).
- **Tiempo de ejecución:** 10 h
- **Servidor:** CloudLab (5 servidores *d6515*)

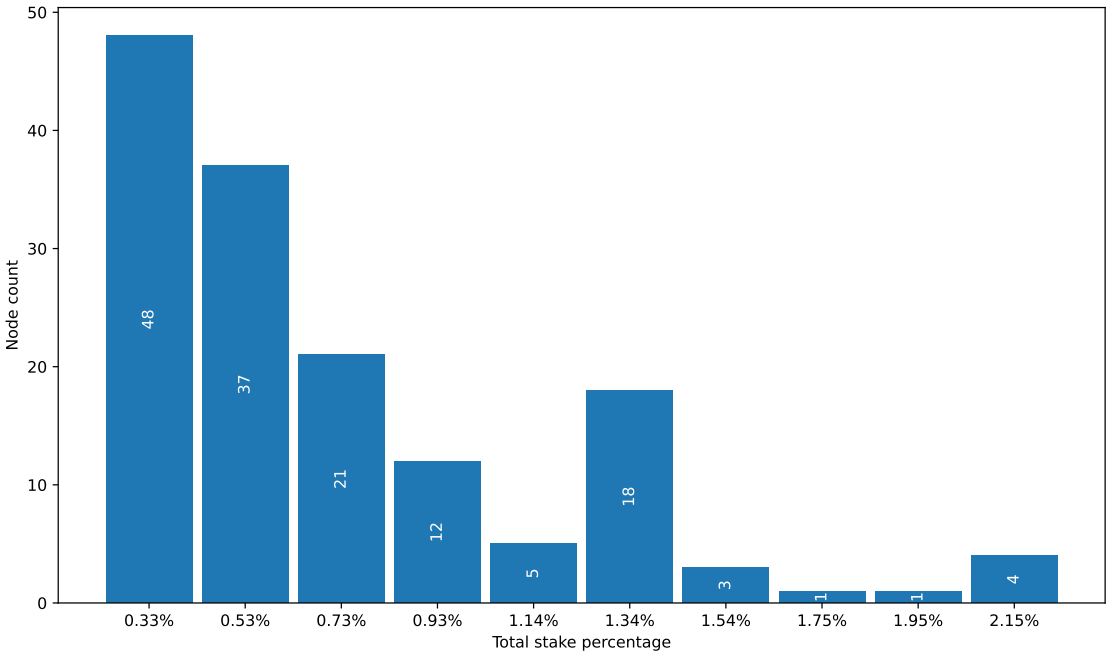


Fig. 4.30: Distribución del *stake* en el sistema según porcentaje del mismo, para 150 nodos y 50 *relays*, con mayoría de *stake* **cercano**. Se marca el valor medio de cada intervalo, con un ancho de *bin* del 0,15 %.

ws(m)	window count	total mins	prc base	rec base	prc count	rec count	prc fibo	rec fibo
1	5	5	1	0.0833	0	0	1	0.1458
1	10	10	0.892	0.4792	0	0	0.7425	0.6042
1	25	25	1	0.6667	0	0	0.8103	0.5833
1	50	50	0.8227	0.8125	0	0	0.6925	0.8125
1	100	100	0.875	0.8125	0	0	0.7304	0.875
1	150	150	0.875	0.8125	0	0	0.75	0.7917
1	200	200	0.5833	0.9375	0	0	0.5833	0.9375
1	250	250	0.875	0.8125	0	0	0.5811	0.9375
1	300	300	0.5833	0.9375	0	0	0.5833	0.9375
1	350	350	0.5833	0.9375	0	0	0.5833	0.9375
1	400	400	0.5833	0.9375	0	0	0.5833	0.9375
1	450	450	0.5833	0.9375	0	0	0.5833	0.9375
1	500	500	0.75	0.875	0	0	0.5833	0.9375
2	5	10	1	0.0833	0	0	0.7927	0.4792
2	10	20	1	0.4167	0	0	0.95	0.3958
2	25	50	1	0.4792	0	0	0.9187	0.4375
2	50	100	1	0.375	0	0	0.8173	0.6875
2	100	200	1	0.4583	0	0	0.7554	0.875
2	150	300	0.8971	0.5208	0	0	0.5833	0.9375
2	200	400	0.875	0.8125	0	0	0.5833	0.9375
2	250	500	1	0.75	0	0	0.5833	0.9375
5	5	25	1	0.3542	0	0	1	0.3958
5	10	50	1	0.2708	0	0	1	0.2708
5	25	125	0.975	0.4167	0	0	0.95	0.5833
5	50	250	1	0.25	0	0	1	0.4167
5	100	500	1	0.25	0	0	1	0.4167
10	5	50	1	0.2083	0	0	1	0.2083
10	10	100	1	0.2917	0	0	1	0.2917
10	25	250	1	0.25	0	0	1	0.25
10	50	500	1	0.25	0	0	1	0.25
20	5	100	1	0.0833	0	0	1	0.0833
20	10	200	1	0.3125	0	0	1	0.3125
20	25	500	1	0.25	0	0	1	0.25
30	5	150	1	0.25	0	0	1	0.25
30	10	300	1	0.2917	0	0	1	0.2917

Tab. 4.19: Precisión y *recall* por estrategia según cantidad y tamaño de ventana para la predicción de cuentas conectadas a los *relays* asociados al *relay* n151, para el sistema de 150 nodos y 50 *relays* con *stake* heterogéneo **cercano**.

En la tabla 4.19 presentamos los resultados de este sistema con *stake* cercano. Al igual que con el sistema anterior, logramos muy buena precisión en muchos de los casos con estrategia base y fibo, pero el *recall* correspondiente es bajo (con alguna excepción donde ambos se encuentran cerca del 80 %). En los casos donde el *recall* es más alto, se perjudica la precisión. Además, vemos que el algoritmo fibo llega a empeorar algunos resultados para ventanas pequeñas, teniendo un impacto positivo solo a partir de ventanas de cinco

minutos.

El mejor resultado lo obtejemos con 100 ventanas de un minuto en el algoritmo base, consiguiendo una precisión del 87 % y *recall* del 81 %. Para la estrategia fibo, el mismo caso logra una precisión del 73 % y *recall* del 87 %. La estrategia count nuevamente falla en todos los casos y no logra hacer predicciones acertadas, repitiendo lo sucedido en el caso cercano del sistema de 60 nodos y 20 *relays*.

Si bien estos resultados no son los mejores, vemos que en comparación con el sistema previo han mejorado las métricas, indicándonos que al tener el *stake* más cerca, es más fácil detectarlo.

Caso semi-realista: *stake* lejano

Continuamos la experimentación entonces alejando el *stake* del *relay* analizado.

Sistema:

- **Nodos:** 150 (n1 a n150)
- ***Relays*:** 50 (n151 a n200)
- ***Relays* conectados por nodo:** 4 lógicos, 4 físicos (con latencia de 20 ms)
- ***Relays* conectados por *relay*:** 4 lógicos, 18 físicos (con latencia de 30 ms)
- **Stake:** heterogéneo (siendo distribuido según la figura 4.31)
- **Distribución de *stake* por *relay*:** incremental desde el *relay* analizado (la mayoría del *stake* se encuentra **lejano** al *relay*).
- **Tiempo de ejecución:** 10 h
- **Servidor:** CloudLab (5 servidores *d6515*)

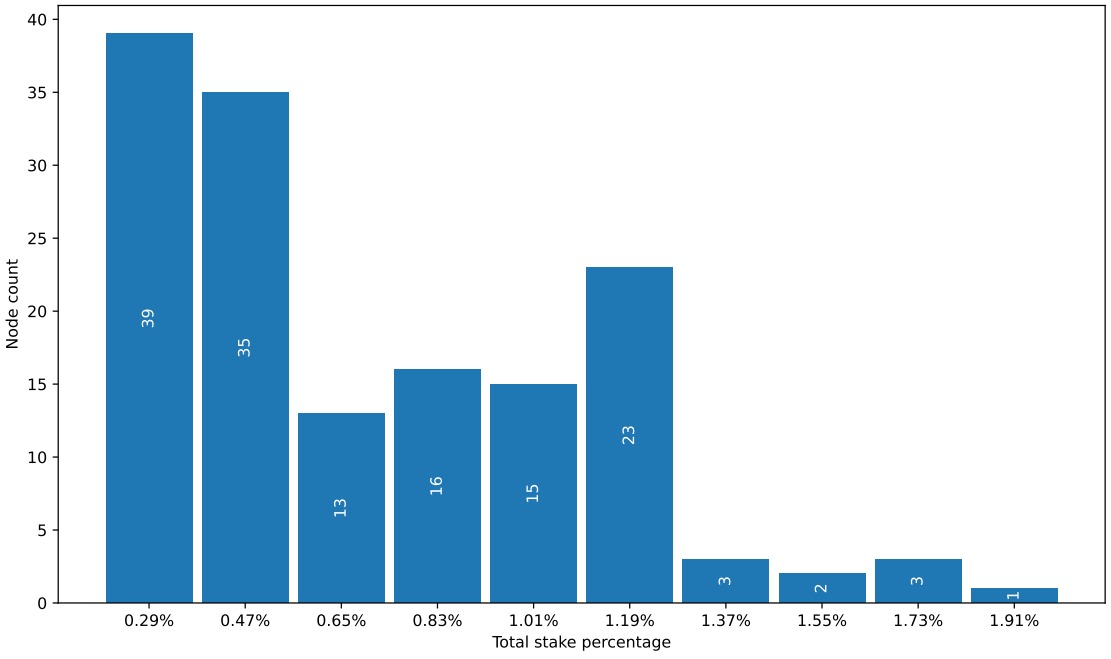


Fig. 4.31: Distribución del *stake* en el sistema según porcentaje del mismo, para 150 nodos y 50 *relays*, con mayoría de *stake* **lejano**. Se marca el valor medio de cada intervalo, con un ancho de *bin* del 0,13 %.

ws(m)	window count	total mins	prc base	rec base	prc count	rec count	prc fibo	rec fibo
1	5	5	0.0432	0.0833	0.125	0.0208	0.5208	0.5417
1	10	10	0.1088	0.2292	0.1519	0.2708	0.425	0.6458
1	25	25	0	0	0.1018	0.1875	0.4027	0.8333
1	50	50	0.0317	0.1875	0.1305	0.3958	0.4242	1
1	100	100	0	0	0.1491	0.6875	0.4103	1
1	150	150	0.0678	0.3958	0.1683	0.6875	0.4103	1
1	200	200	0	0	0.1832	0.875	0.4103	1
1	250	250	0.0354	0.25	0.1703	0.7917	0.4103	1
1	300	300	0.0313	0.25	0.1386	0.4792	0.4103	1
1	350	350	0.0313	0.25	0.1263	0.625	0.4103	1
1	400	400	0.0313	0.25	0.1351	0.6875	0.4103	1
1	450	450	0.0313	0.25	0.1346	0.6667	0.4103	1
1	500	500	0.0313	0.25	0.1244	0.7083	0.4103	1
2	5	10	0.025	0.0417	0.0294	0.0417	0.1424	0.1458
2	10	20	0.0147	0.0208	0.0633	0.1042	0.4776	0.75
2	25	50	0.0172	0.0417	0.1704	0.2917	0.5876	0.7917
2	50	100	0	0	0	0	0.5822	0.75
2	100	200	0	0	0.0609	0.1458	0.4103	1
2	150	300	0.0052	0.0208	0.083	0.2917	0.4103	1
2	200	400	0	0	0.0713	0.2292	0.4103	1
2	250	500	0.0053	0.0208	0.0742	0.25	0.4103	1
5	5	25	0	0	0	0	0.5208	0.6042
5	10	50	0	0	0.0444	0.0625	0.4829	0.5
5	25	125	0	0	0	0	0.5898	0.7292
5	50	250	0	0	0	0	0.6015	0.7708
5	100	500	0	0	0	0	0.563	0.7708
10	5	50	0	0	0	0	0.4718	0.5833
10	10	100	0	0	0	0	0.7236	0.6875
10	25	250	0	0	0	0	0.605	0.75
10	50	500	0	0	0	0	0.6015	0.7708
20	5	100	0	0	0	0	0.675	0.5208
20	10	200	0	0	0	0	0.4777	0.4583
20	25	500	0	0	0	0	0.354	0.6667
30	5	150	0	0	0	0	0.4306	0.125
30	10	300	0	0	0	0	0.625	0.6458

Tab. 4.20: Precisión y *recall* por estrategia según cantidad y tamaño de ventana para la predicción de cuentas conectadas a los *relays* asociados al *relay* n151, para el sistema de 150 nodos y 50 *relays* con *stake* heterogéneo **lejano**.

En la tabla 4.20 encontramos los resultados para este sistema, donde inmediatamente notamos una disminución importante en las métricas de todas las estrategias. Para la base, no se logra superar el 10 % de precisión con *recall* de 22 % (excepto casos donde mejora esta segunda métrica en detrimento de la primera).

La estrategia count mejora los resultados de la base, pero tampoco logra buenos resultados. Para ventanas de un minuto logra precisión del 18 % con *recall* del 87 %, mejorando

sus resultados para el caso cercano, pero sin mejorar contra el primer sistema de *stake* más equitativo.

Finalmente, la estrategia fibo mejora notablemente lo obtenido por la base, logrando como mejor resultado un 72 % de precisión y 68 % de *recall* para diez ventanas de diez minutos. También logra valores de *recall* mucho más altos, pero afectando la precisión.

Para esta estrategia, los resultados obtenidos son peores que en el caso cercano: notamos nuevamente la dificultad en la predicción al alejar los nodos con mayor cantidad de *stake*. También notamos que, en comparación con el caso del *stake* distribuído de forma más equitativa, los resultados son menos precisos y agregamos falsos positivos a nuestras predicciones. Estos resultados nos podrían revelar que este escenario es el peor caso para las predicciones. Buscamos entonces ver qué sucedería en una red de escala real.

4.7.8. Caso realista

Para completar los experimentos, realizamos una experiencia realista con 300 nodos y 100 *relays* con *stake* heterogéneo para comparar con los resultados obtenidos previamente con las redes más pequeñas y la distribución homogénea.

Sistema:

- **Nodos:** 300 (n1 a n300)
- ***Relays*:** 100 (n301 a n400)
- ***Relays* conectados por nodo:** 4 lógicos, 4 físicos (con latencia de 20 ms)
- ***Relays* conectados por *relay*:** 4 lógicos, 46 físicos (con latencia de 30 ms)
- **Stake:** heterogéneo (siendo distribuido según la figura 4.32)
- **Promedio de *stake* por *relay*:** 1 % (*RSD* 20 %)
- **Tiempo de ejecución:** 10 h
- **Servidor:** CloudLab (20 servidores *d6515*)

En este experimento vemos en la tabla 4.21 un resultado que sigue la línea de lo obtenido previamente: al incrementar el tamaño de la red, aumenta la dificultad para predecir y nuestra precisión cae fuertemente. En algunos casos puntuales logramos precisión de más del 80 %, pero siempre con *recall* muy bajo para estos casos (menos del 47 %). La precisión más alta la encontramos para 100 ventanas de cinco minutos, con un valor del 93 %, mientras que su *recall* es casi del 20 %.

Con estos resultados notamos que podemos tener resultados decentes de precisión (superando el 80 %), pero nuestro *recall* siempre resulta bajo. No estamos reconociendo la totalidad de las cuentas detectadas al *relay* analizado.

Replicamos lo experimentado en los otros tamaños con sistemas donde distribuímos el *stake* de la red primero en forma cercana al *relay* analizado, y luego de manera lejana.

Caso realista: *stake* cercano

Acercamos el *stake* a las cuentas conectadas a los *relays* vecinos del analizado para experimentar con una red de escala realista.

Sistema:

- **Nodos:** 300 (n1 a n300)
- ***Relays*:** 100 (n301 a n400)
- ***Relays* conectados por nodo:** 4 lógicos, 4 físicos (con latencia de 20 ms)
- ***Relays* conectados por *relay*:** 4 lógicos, 46 físicos (con latencia de 30 ms)
- **Stake:** heterogéneo (siendo distribuido según la figura 4.33)

- **Distribución de *stake* por *relay*:** decremental desde el *relay* analizado (la mayoría del *stake* se encuentra **cercano** al *relay*).
- **Tiempo de ejecución:** 10 h
- **Servidor:** CloudLab (20 servidores *d6515*)

En este experimento obtenemos los resultados de la tabla 4.22, donde vemos buenas predicciones no solo para el algoritmo fibo, sino también para el base. La estrategia count vuelve a fallar en otro caso de *stake* cercano sin lograr tener una sola predicción.

La estrategia fibo en este caso logra mejorar lo obtenido por la base en ventanas de más de cinco minutos, ampliando el *recall*, pero mantieniendo resultados muy similares. En ventanas más chicas, vemos que llega hasta a empeorar las métricas base, pero son pocos los casos donde esta disminución en las mismas sea sumamente significativa.

Para ambas estrategias, logramos predicciones de más del 80 % de precisión y *recall*: por ejemplo, con 25 ventanas de diez minutos, superan el 82 % de precisión y logran 100 % de *recall*.

Estos resultados son sumamente superiores a lo obtenido en el caso anterior, donde la distribución del *stake* se encuentra más equitativa en la red. El hecho de que los nodos de mayor participación se encuentren directamente conectados a los *relays* vecinos nos ayuda a ser mejores prediciendo.

Caso realista: *stake* lejano

Revisamos entonces qué sucede si alejamos nuevamente a los nodos de mayor participación.

Sistema:

- **Nodos:** 300 (n1 a n300)

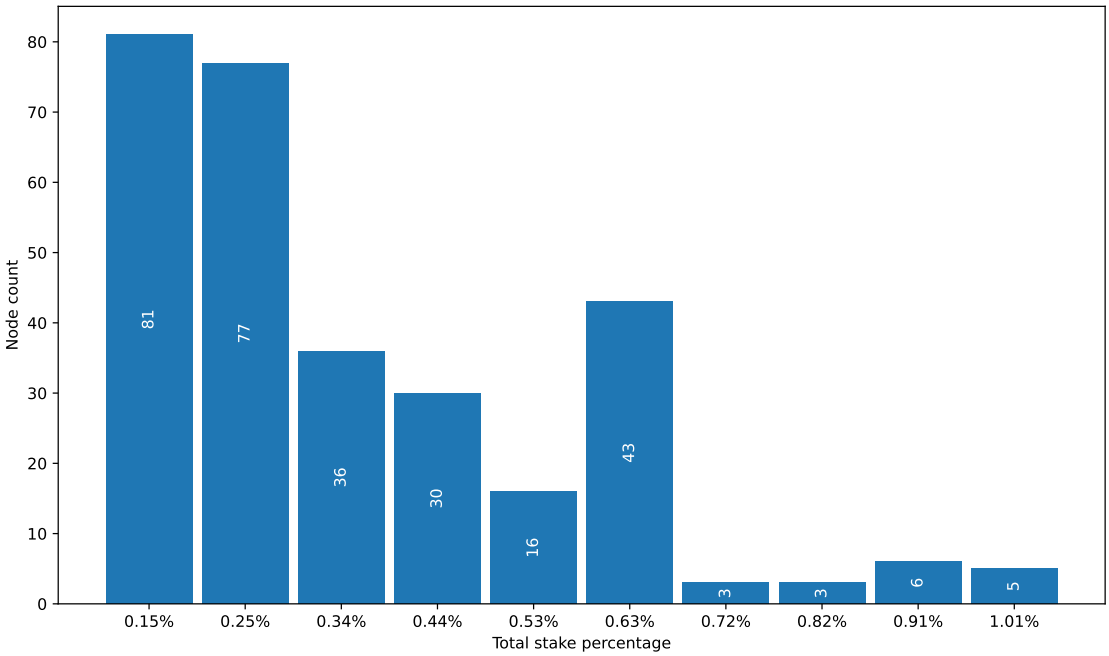


Fig. 4.32: Distribución del *stake* en el sistema según porcentaje del mismo, para 300 nodos y 100 *relays*. Se marca el valor medio de cada intervalo, con un ancho de bin del 0,05 %.

ws(m)	window count	total mins	prc base	rec base	prc count	rec count	prc fibo	rec fibo
1	5	5	0.2436	0.2778	0.134	0.1944	0.2439	0.25
1	10	10	0.4848	0.1667	0.1674	0.1667	0.4739	0.3333
1	25	25	0.2222	0.1111	0.0648	0.2222	0.3258	0.2778
1	50	50	0.293	0.2222	0.0612	0.2222	0.2662	0.4722
1	100	100	0.3333	0.1944	0.1397	0.3056	0.6389	0.3611
1	150	150	0.3542	0.3889	0.0917	0.3056	0.3819	0.5556
1	200	200	0.5778	0.2778	0.0941	0.3056	0.8462	0.25
1	250	250	0.4444	0.3611	0.0914	0.3056	0.5167	0.5278
1	300	300	0.4444	0.3611	0.0902	0.3056	0.5673	0.4722
1	350	350	0.6333	0.2222	0.0897	0.3056	0.6458	0.3333
1	400	400	0.5778	0.2778	0.0897	0.3056	0.6458	0.3333
1	450	450	0.5778	0.2778	0.0897	0.3056	0.6429	0.3056
1	500	500	0.4111	0.3333	0.0902	0.3056	0.8125	0.25
1	600	600	0.5778	0.2778	0.0902	0.3056	0.8125	0.25
2	5	10	0.5	0.1111	0.1462	0.1389	0.55	0.2778
2	10	20	0.4444	0.1389	0.1381	0.2778	0.373	0.3333
2	25	50	0.2917	0.1111	0.2496	0.25	0.6012	0.1944
2	50	100	0.4762	0.1667	0.1423	0.25	0.6204	0.25
2	100	200	0.5	0.1111	0.1064	0.3056	0.7667	0.25
2	150	300	0.6667	0.1389	0.1	0.3056	0.5613	0.3611
2	200	400	0.5	0.1111	0.096	0.3056	0.5683	0.3611
2	250	500	0.6111	0.1111	0.0965	0.3056	0.5724	0.3889
5	5	25	0.5	0.0556	0.1604	0.1944	0.6768	0.25
5	10	50	0.3333	0.0556	0.1259	0.1944	0.1667	0.0278
5	25	125	0.6111	0.0833	0.1441	0.25	0.5016	0.2778
5	50	250	0.5	0.1111	0.1049	0.3056	0.5139	0.25
5	100	500	0.5	0.1111	0.1036	0.3056	0.9333	0.1944
10	5	50	0.4444	0.1111	0.1389	0.1389	0.8333	0.1944
10	10	100	0.1667	0.0278	0.1474	0.1944	0.5833	0.2222
10	25	250	0.5	0.1111	0.1664	0.25	0.7667	0.1667
10	50	500	0.3889	0.1389	0.1099	0.3056	0.5444	0.2222
20	5	100	0.6111	0.1111	0.3529	0.0556	0.6111	0.1111
20	10	200	0.4444	0.0833	0.1711	0.2222	0.6556	0.2222
20	25	500	0.6111	0.0833	0.1339	0.3056	0.7111	0.1667
30	5	150	0.4444	0.1111	0.2444	0.1389	0.5278	0.1667
30	10	300	0.4444	0.0833	0.3554	0.25	0.8222	0.1944

Tab. 4.21: Precisión y *recall* por estrategia según cantidad y tamaño de ventana para la predicción de cuentas conectadas a los *relays* asociados al *relay* n301, para el sistema de 300 nodos y 100 *relays* con *stake* heterogéneo.

- ***Relays*:** 100 (n301 a n400)
- ***Relays* conectados por nodo:** 4 lógicos, 4 físicos (con latencia de 20 ms)
- ***Relays* conectados por *relay*:** 4 lógicos, 46 físicos (con latencia de 30 ms)
- **Stake:** heterogéneo (siendo distribuido según la figura 4.34)
- **Distribución de *stake* por *relay*:** incremental desde el *relay* analizado (la mayoría

del *stake* se encuentra **lejano** al *relay*).

- **Tiempo de ejecución:** 10 h
- **Servidor:** CloudLab (20 servidores *d6515*)

Para este escenario con *stake* lejano, obtenemos los resultados de la tabla 4.23. En todas las estrategias vemos valores sumamente bajos, sobre todo en comparación con el escenario cercano, pero también con el escenario con *stake* más equitativo.

Las estrategias base y count no superan el 3 % de precisión, con valores de *recall* bajos. La estrategia fibo, si bien mejora ambas métricas para todos los casos, tampoco logra superar el 23 % de precisión, pero logrando valores muy buenos de *recall*. Es decir, predecimos como conectados todos aquellos que lo están, pero también un grupo grande que no (agregando así falsos positivos y disminuyendo la precisión).

Este resultado refleja el peor caso para nuestras predicciones: no podemos predecir con precisión cuáles son las cuentas que están conectadas a los *relays* vecinos.

Sin embargo, viendo los resultados del caso donde el *stake* se encuentra cercano al *relay* y el anterior donde se encuentra distribuído equitativamente en la red, notamos buenos valores de precisión en nuestras estrategias base y fibo. Nos surge cuestionarnos qué conjunto de las cuentas conectadas son las que sí estamos reconociendo como conectadas correctamente.

4.8. Stake heterogéneo: reconociendo el top cinco

Para un atacante, en una red con *stake* heterogéneo, el interés central va a ser encontrar dónde están las cuentas con mayor cantidad de *stake* en el sistema. Si bien los resultados

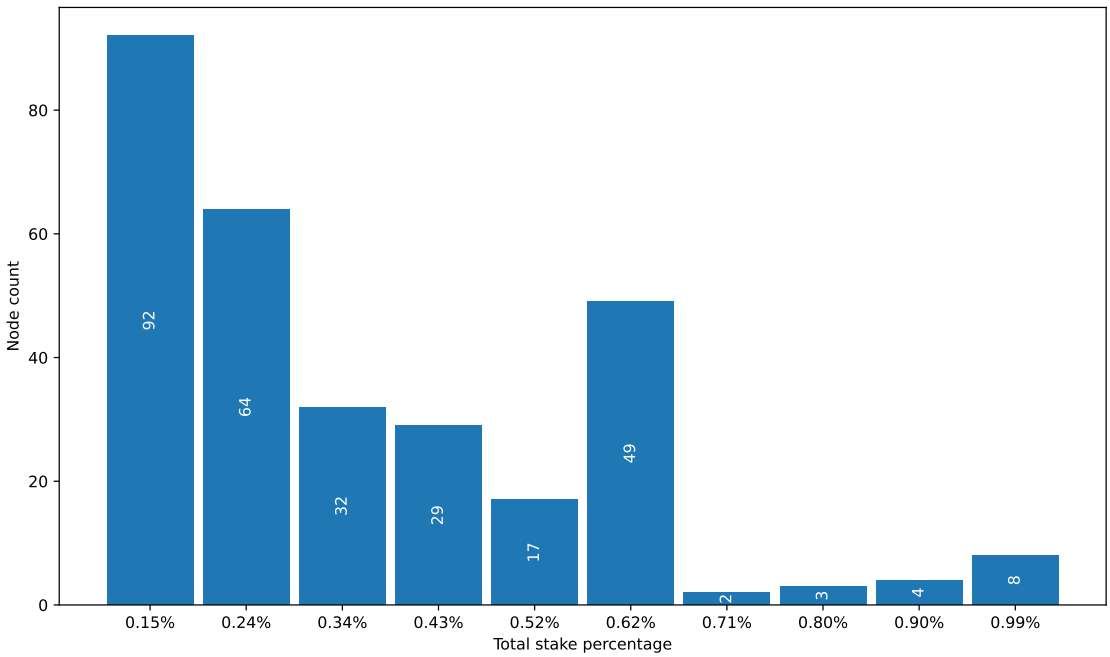


Fig. 4.33: Distribución del *stake* en el sistema según porcentaje del mismo, para 300 nodos y 100 *relays*, con mayoría de *stake* **cercano**. Se marca el valor medio de cada intervalo, con un ancho de bin del 0,05 %.

ws(m)	window count	total mins	prc base	rec base	prc count	rec count	prc fibo	rec fibo
1	5	5	0	0	0	0	0	0
1	10	10	0	0	0	0	0	0
1	25	25	0.0833	0.0208	0	0	0	0
1	50	50	0.0833	0.0208	0	0	0	0
1	100	100	0	0	0	0	0	0
1	150	150	0.1154	0.125	0	0	0	0
1	200	200	0.125	0.1042	0	0	0	0
1	250	250	0.7794	0.6875	0	0	0.4636	0.7083
1	300	300	0.1591	0.1458	0	0	0	0
1	350	350	0.6146	0.7917	0	0	0.5917	0.6667
1	400	400	0.8034	0.8333	0	0	0.7461	0.5417
1	450	450	0.8558	0.7292	0	0	0.7235	0.5
1	500	500	0.8379	0.7917	0	0	0.6959	0.8125
2	5	10	0.875	0.3125	0	0	0.625	0.25
2	10	20	0.7112	0.3542	0	0	0.6034	0.3333
2	25	50	0.7117	0.7708	0	0	0.652	0.5208
2	50	100	0.7284	0.5625	0	0	0.641	0.5625
2	100	200	0.8558	0.6458	0	0	0.6907	0.8333
2	150	300	0.6973	0.8958	0	0	0.6274	0.8125
2	200	400	0.8379	0.9375	0	0	0.7	0.9583
2	250	500	0.8245	0.75	0	0	0.7174	0.9792
5	5	25	0.8429	0.3125	0	0	0.8429	0.3125
5	10	50	0.8542	0.7292	0	0	0.8363	0.8542
5	25	125	0.8314	0.8958	0	0	0.8083	0.9167
5	50	250	0.8269	1	0	0	0.754	1
5	100	500	0.8419	1	0	0	0.8062	1
10	5	50	1	0.3333	0	0	0.8542	0.3542
10	10	100	0.9167	0.7083	0	0	0.875	0.7292
10	25	250	0.8462	1	0	0	0.8297	1
10	50	500	0.8462	1	0	0	0.8462	1
20	5	100	0.9375	0.4583	0	0	0.875	0.6458
20	10	200	0.9091	0.6042	0	0	0.75	0.6667
20	25	500	0.875	1	0	0	0.8696	0.9583
30	5	150	1	0.2083	0	0	1	0.5208
30	10	300	0.8906	0.8542	0	0	0.75	0.6667

Tab. 4.22: Precisión y *recall* por estrategia según cantidad y tamaño de ventana para la predicción de cuentas conectadas a los *relays* asociados al *relay* n301, para el sistema de 300 nodos y 100 *relays* con *stake* heterogéneo **cercano**.

anteriores no resultan significativos para poder reconstruir toda la red, resultaría muy interesante si podemos identificar a las cuentas más importantes del sistema (y no preocuparnos por las de menor *stake*, ya que su participación es mucho menos significativa).

Revisando los resultados anteriores, pudimos ver que en los casos con mayor precisión, logramos detectar las cuentas con mayor *stake* de las conectadas a cada uno de los *relays*, de manera que planteamos dos nuevos experimentos para poder medir y analizar qué tan

buenos somos para encontrar estas cuentas.

Planteamos un sistema con 300 nodos y 100 *relays* donde vamos a conectar las cinco cuentas con mayor cantidad de *stake* a *relays* que se encuentren conectados a nuestro *relay* víctima, y nuestro objetivo es ver si podemos detectarlas como conectadas a los mismos en un escenario similar al visto previamente. Luego, para confirmar que no se trate de una casualidad, replicamos el mismo escenario, pero desconectando esas cuentas de los *relays* y viendo que no las detectamos como conectadas.

La elección de cinco cuentas se debe a que esa cantidad representa para la red actual el 20 % del *stake* participante del consenso. Es decir, afectando esta cantidad de cuentas nos alcanzaría para poder detener la red real según lo visto en la primer parte de este trabajo.

Con estos escenarios, probamos primero nuestra habilidad para encontrar el top cinco si está conectado a nuestros *relays*, y luego verificar que si no lo estuvieran, no las detectaríamos como conectadas.

4.8.1. Top cinco conectado

En este escenario contamos con 300 nodos y 100 *relays*, entre ellos el *relay* n301 que se conecta a cuatro otros *relays* del sistema. Distribuimos las cinco cuentas con mayor cantidad de *stake* de manera que estén conectadas cada una a uno de los *relays* con los que se comunica el n301. Nos interesa analizar si detectamos como conectada cada una de las cuentas al *relay* correspondiente.

Ahora nuestra precisión va a medir la cantidad de cuentas del top cinco que identificamos conectadas al *relay* correcto, sobre la totalidad de las cuentas del top cinco que detectamos conectadas (sea porque realmente están conectadas, o si erróneamente decidi-

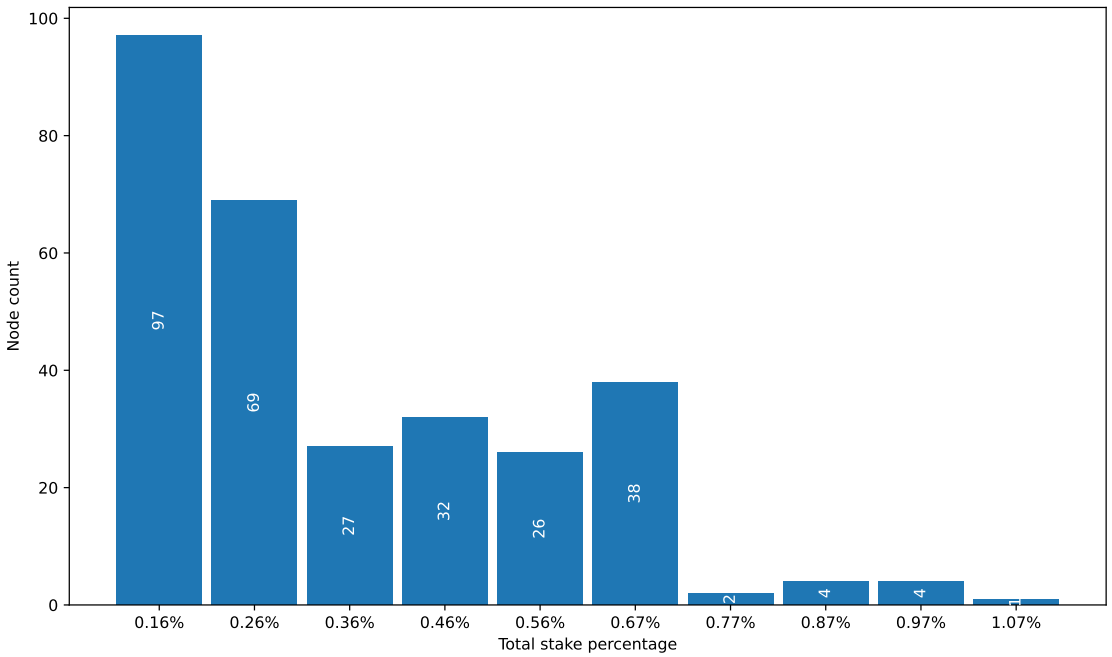


Fig. 4.34: Distribución del *stake* en el sistema según porcentaje del mismo, para 300 nodos y 100 *relays*, con mayoría de *stake* **lejano**. Se marca el valor medio de cada intervalo, con un ancho de bin del 0,05 %.

ws(m)	window count	total mins	prc base	rec base	prc count	rec count	prc fibo	rec fibo
1	5	5	0	0	0	0	0.0778	0.0625
1	10	10	0	0	0	0	0.2306	0.1667
1	25	25	0.0093	0.0417	0.0098	0.0417	0.0078	0.0208
1	50	50	0.0063	0.0417	0.0121	0.0833	0.0798	0.4583
1	100	100	0.0074	0.0625	0.0173	0.1875	0.1343	0.9792
1	150	150	0	0	0.0174	0.1667	0.1362	1
1	200	200	0.0049	0.0625	0.0299	0.3542	0.1346	1
1	250	250	0	0	0.0264	0.3125	0.1339	1
1	300	300	0	0	0.0202	0.25	0.1338	1
1	350	350	0	0	0.0264	0.3958	0.1328	1
1	400	400	0.0026	0.0208	0.0301	0.4375	0.1328	1
1	450	450	0.0025	0.0208	0.0263	0.3333	0.1345	1
1	500	500	0.0025	0.0208	0.0212	0.2708	0.1345	1
2	5	10	0	0	0	0	0.2222	0.2917
2	10	20	0	0	0	0	0.1439	0.4375
2	25	50	0	0	0	0	0.1048	0.4583
2	50	100	0	0	0	0	0.1434	0.9792
2	100	200	0	0	0	0	0.1393	1
2	150	300	0	0	0	0	0.1397	1
2	200	400	0	0	0	0	0.1392	1
2	250	500	0	0	0	0	0.1392	1
5	5	25	0	0	0	0	0.1967	0.2292
5	10	50	0	0	0	0	0.1071	0.2917
5	25	125	0	0	0	0	0.1387	0.625
5	50	250	0	0	0	0	0.1551	0.9375
5	100	500	0	0	0	0	0.1461	1
10	5	50	0	0	0	0	0.1183	0.1667
10	10	100	0	0	0	0	0.1359	0.5
10	25	250	0	0	0	0	0.1697	0.8333
10	50	500	0	0	0	0	0.16	0.9792
20	5	100	0	0	0	0	0.14	0.2083
20	10	200	0	0	0	0	0.1486	0.3333
20	25	500	0	0	0	0	0.1324	0.5833
30	5	150	0	0	0	0	0.1383	0.1458
30	10	300	0	0	0	0	0.181	0.3125

Tab. 4.23: Precisión y *recall* por estrategia según cantidad y tamaño de ventana para la predicción de cuentas conectadas a los *relays* asociados al *relay* n301, para el sistema de 300 nodos y 100 *relays* con *stake* heterogéneo **lejano**.

mos que lo estaban). El *recall* nos va a indicar qué proporción de las cuentas del top cinco estamos detectando como conectadas del total de ellas (cinco).

Vamos a considerar como *true positive* cuando detectemos conectada una de las cuentas del top cinco al *relay* que le corresponda, considerando como un error si la detectamos a un *relay* que no le corresponda (en ese caso será un *false positive*). De esta manera, no solo nos interesa detectar las cuentas, sino hacerlo para el *relay* correcto.

Ejecutamos una corrida de diez horas como en los casos anteriores y analizamos los resultados midiendo estas nuevas métricas para las estrategias propuestas.

Sistema:

- **Nodos:** 300 (n1 a n300)
- ***Relays*:** 100 (n301 a n400)
- ***Relays* conectados por nodo:** 4 lógicos, 4 físicos (con latencia de 20 ms)
- ***Relays* conectados por *relay*:** 4 lógicos, 46 físicos (con latencia de 30 ms)
- ***Stake*:** heterogéneo (siendo distribuido según la figura 4.35)
- **Tiempo de ejecución:** 10 h
- **Servidor:** CloudLab (20 servidores *d6515*)

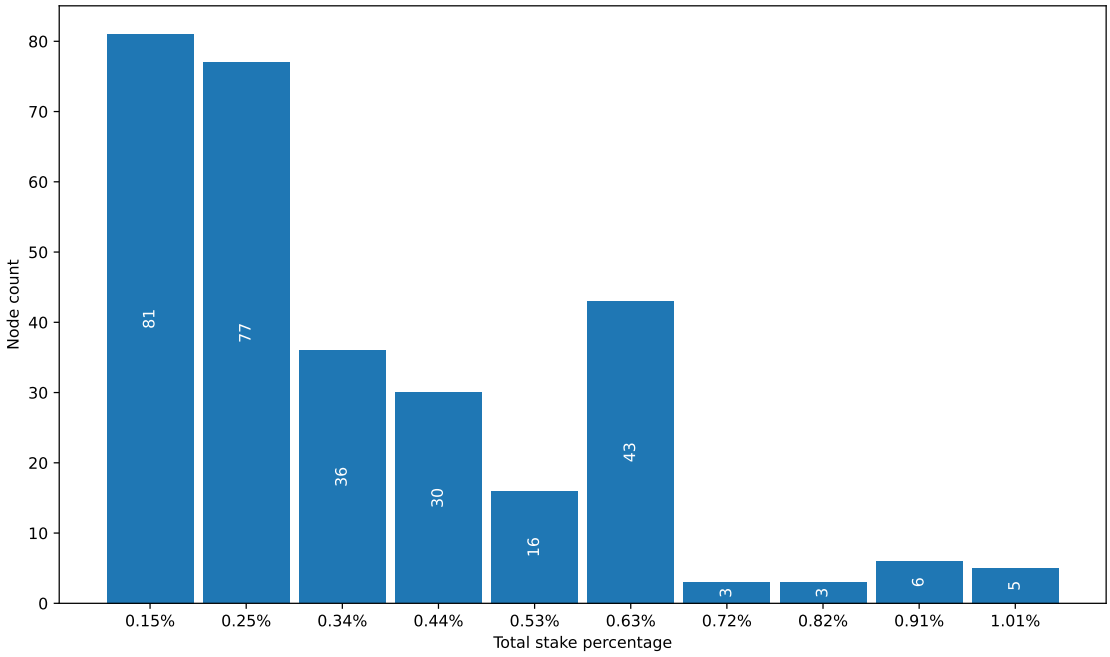


Fig. 4.35: Distribución del *stake* en el sistema según porcentaje del mismo, para 300 nodos y 100 *relays*. Se marca el valor medio de cada intervalo, con un ancho de bin del 0,05 %.

Basándonos en los resultados de la tabla 4.24, la estrategia de *count* demuestra nuevamente no ser buena, pero vemos que tanto la estrategia base y la *fibo* obtienen muy buenos resultados para detectar estas cuentas, encontrando a la gran mayoría en muchos casos. El mejor resultado lo encontramos en ventanas pequeñas de dos minutos, tomando cinco de ellas: con esfuerzo de diez minutos logramos detectar correctamente todas las cuentas del top cinco que están conectadas a los *relays* vecinos. Luego, con ventanas de un minuto también logramos buenos resultados consistentemente tomando entre 25 y 150 ventanas (para el caso de la estrategia *fibo*).

En comparación, la estrategia *fibo* no plantea una mejora notable contra la estrategia base. Esto se debe a que, como vimos previamente, la estrategia *fibo* mejora la distinción entre nodos no conectados de los conectados para un *relay* cerca del salto que distingue a los dos, incrementando la diferencia para nodos con menor *stake*. Luego, si los nodos del top cinco se encuentran hacia la derecha del gráfico (es decir, luego del salto), la ponderación que realizamos no va a mejorar su detección. Por este motivo los resultados

ws(m)	window count	total mins	prc base	rec base	prc count	rec count	prc fibo	rec fibo
1	5	5	1	0.8	0	0	1	0.6
1	10	10	1	0.8	0	0	1	0.8
1	25	25	1	1	0	0	1	1
1	50	50	1	1	0	0	1	1
1	100	100	1	0.8	0	0	1	1
1	150	150	1	0.8	0	0	1	1
1	200	200	1	0.8	0	0	1	0.6
1	250	250	1	1	0	0	1	1
1	300	300	1	1	0	0	1	0.8
1	350	350	1	1	0	0	1	0.8
1	400	400	1	0.8	0	0	1	0.8
1	450	450	1	0.8	0	0	1	0.8
1	500	500	1	0.8	0	0	1	0.8
1	600	600	1	0.8	0	0	1	0.8
2	5	10	1	1	0	0	1	1
2	10	20	1	0.8	0	0	1	0.8
2	25	50	1	0.8	0	0	1	0.8
2	50	100	1	0.8	0	0	1	0.8
2	100	200	1	0.8	0	0	1	0.8
2	150	300	1	0.8	0	0	1	0.8
2	200	400	1	0.8	0	0	1	0.8
2	250	500	1	0.8	0	0	1	0.8
5	5	25	1	0.8	0	0	1	0.8
5	10	50	1	0.8	0	0	1	0.8
5	25	125	1	0.8	0	0	1	1
5	50	250	1	0.8	0	0	1	0.8
5	100	500	1	0.8	0	0	1	0.8
10	5	50	1	0.8	0	0	1	0.8
10	10	100	1	1	0	0	1	1
10	25	250	1	0.8	0	0	1	0.6
10	50	500	1	1	0	0	1	0.6
20	5	100	1	0.8	0	0	1	0.8
20	10	200	1	0.8	0	0	1	0.8
20	25	500	1	0.8	0	0	1	0.6
30	5	150	1	0.6	0	0	1	0.8
30	10	300	1	0.8	0	0	1	0.8

Tab. 4.24: Precisión y *recall* por estrategia según cantidad y tamaño de ventana para la predicción de conexión del top 5 del *stake* del sistema conectado a los *relays* asociados al *relay* n301, para el sistema de 300 nodos y 100 *relays* con *stake* heterogéneo.

son similares para ambas opciones.

Considerando los valores obtenidos, resulta plausible poder establecer una estrategia para predecir si alguna de las cuentas con mayor cantidad de *stake* se encuentra conectada a alguno de los *relays* vecinos y a cuál de ellos.

Para confirmar que sea así, planteamos otro experimento donde se mantiene la misma

estructura de la red, pero sin conectar el top cinco a los *relays* vecinos.

4.8.2. Top cinco no conectado

En esta red, el *relay* n301 se conecta a cuatro *relays* que no contienen a ninguna de las cuentas del top cinco de *stake* del sistema. Por ende, nuestro objetivo es verificar que con la información capturada no detectemos erróneamente como conectado a alguno de estas cuentas en los *relays* analizados.

El sistema y distribución de *stake* son los mismos que en el caso anterior, solo se modifica la conexión de las cuentas del top cinco y se elimina la conexión a estos *relays*.

Entonces, nuestro experimento es analizado considerando como precisión los verdaderos positivos, la cantidad de cuentas del top cinco no conectadas que detectamos como no conectadas (de 0 a 5) sobre la cantidad total de cuentas del top cinco detectadas como no conectadas. Pero dado que en nuestro experimento la totalidad están no conectadas, resulta trivial que la precisión es 1. Luego, nos interesa conocer el *recall* de nuestro experimento: cuántas de estas cuentas no conectadas estamos detectando como no conectadas. Esta métrica nos indica qué porcentaje de las cuentas no conectadas estamos prediciendo correctamente como no conectadas. Este valor solo podría decrementar si en algún caso predecimos una de ellas como conectada erróneamente.

El resultado mostrado en la tabla 4.25, si bien puede parecer simple, nos indica que para todos los algoritmos la detección de las cuentas del top 5 como no conectadas es perfecta. Si bien para el caso de la estrategia *count* se debe a que es mala y marca como no conectadas a todas las cuentas, en el caso de las otras estrategias donde sí tenemos una buena detección en el caso donde están conectadas nos da una pauta de que no nos dan información falsa si no lo estuvieran.

Así, si nuestro *relay* se conectara a otro *relay*, podríamos predecir con un gran grado de certidumbre si alguna de las cuentas con mayor cantidad de *stake* de la red se encuentra conectada a uno de ellos, y también sabríamos con excelente certeza que si no está conectada, no la predecimos como conectada (evitando falsos positivos).

ws(m)	window count	total mins	prc base	rec base	prc count	rec count	prc fibo	rec fibo
1	5	5	1	1	1	1	1	1
1	10	10	1	1	1	1	1	1
1	25	25	1	1	1	1	1	1
1	50	50	1	1	1	1	1	1
1	100	100	1	1	1	1	1	1
1	150	150	1	1	1	1	1	1
1	200	200	1	1	1	1	1	1
1	250	250	1	1	1	1	1	1
1	300	300	1	1	1	1	1	1
1	350	350	1	1	1	1	1	1
1	400	400	1	1	1	1	1	1
1	450	450	1	1	1	1	1	1
1	500	500	1	1	1	1	1	1
1	600	600	1	1	1	1	1	1
2	5	10	1	1	1	1	1	1
2	10	20	1	1	1	1	1	1
2	25	50	1	1	1	1	1	1
2	50	100	1	1	1	1	1	1
2	100	200	1	1	1	1	1	1
2	150	300	1	1	1	1	1	1
2	200	400	1	1	1	1	1	1
2	250	500	1	1	1	1	1	1
5	5	25	1	1	1	1	1	1
5	10	50	1	1	1	1	1	1
5	25	125	1	1	1	1	1	1
5	50	250	1	1	1	1	1	1
5	100	500	1	1	1	1	1	1
10	5	50	1	1	1	1	1	1
10	10	100	1	1	1	1	1	1
10	25	250	1	1	1	1	1	1
10	50	500	1	1	1	1	1	1
20	5	100	1	1	1	1	1	1
20	10	200	1	1	1	1	1	1
20	25	500	1	1	1	1	1	1
30	5	150	1	1	1	1	1	1
30	10	300	1	1	1	1	1	1

Tab. 4.25: Precisión y *recall* por estrategia según cantidad y tamaño de ventana para la predicción de conexión del top 5 del *stake* del sistema no conectado a los *relays* asociados al *relay* n301, para el sistema de 300 nodos y 100 *relays* con *stake* heterogéneo.

5. CONCLUSIONES Y TRABAJO FUTURO

5.1. Conclusiones generales

Como objetivo de la tesis nos propusimos analizar como se comporta una red de *Proof of Stake* ante diferentes alteraciones en su topología física y, en base a esos resultados, experimentar con formas de predecir cuál es la topología lógica, ejecutando pruebas en ambientes totalmente controlados y privados (evitando así cualquier disrupción que pudiera surgir de la imprevisibilidad de las conexiones por Internet). También nos interesó analizar si estas alteraciones podrían ser objetivo de un ataque malicioso, qué esfuerzo conllevaría para un potencial atacante y cuál es el impacto del mismo.

Para esto tomamos como ejemplo de red de *PoS* a `Algorand`, que plantea ideas innovadoras en cuanto a la resolución de protocolo de consenso descentralizado al utilizar una función criptográfica que puede ser ejecutada de manera *offline* para la toma de decisiones. Además, la fundación `Algorand` se encuentra trabajando de manera cercana con las instituciones académicas, siendo de gran interés el enfoque de investigación con la misma.

Otro factor clave para este trabajo era la posibilidad de ejecutar múltiples instancias de un cliente de la red, intentando alcanzar niveles realistas de nodos que replicaran las cantidades actuales de la red elegida. Esto lo logramos utilizando `SherlockFog` y Docker: `SherlockFog` nos proveyó la flexibilidad para construir la red física, configurar los *delays* en los enlaces entre los nodos, correr una cantidad dada de nodos para la imagen Docker y realizar experimentación con los mismos, corriendo todo en múltiples servidores para poder balancear el consumo de recursos.

Con estos objetivos implementamos cambios en el cliente de `Algorand` para poder registrar los eventos que nos resultaban de interés, sin hacer modificaciones sustanciales al mismo, y generamos una imagen `docker` conteniendo al mismo y *script*s para configurarlo y correrlo externamente.

Como queríamos poder reproducir los experimentos de este trabajo, buscamos una manera de crear todos los nodos de la red con anterioridad a su ejecución. Con esta creación, configuramos no solo a cada nodo con sus características propias, sino todas las conexiones lógicas que utiliza durante el experimento. Así, no existe aleatoriedad en cómo se conecta la red y descartamos un grado de incertidumbre en la experimentación.

Una vez implementada la imagen Docker y creados los nodos, utilizamos `SherlockFog` para emular la topología física sobre la cual luego construiríamos la red lógica de `Algorand`, ejecutando múltiples instancias de clientes equitativamente distribuidos en los servidores necesarios para cada experimento. Dependiendo del tamaño de cada experimento, utilizamos un entorno local o múltiples servidores en CloudLab para lograr tiempos normales de ronda.

En primera instancia experimentamos alterando la topología física sobre la que corren los clientes de `Algorand`. Comenzamos con una red de diez nodos y diez *relays* con *stake* homogéneo (con un nodo conectado a cada *relay*), donde los *relays* se conectan físicamente entre sí formando una línea. Interrumpimos esta cadena por la mitad y vimos que la producción de bloques se detiene por completo en ambos extremos cuando se produce el corte. Apenas se levanta, la red reanuda su avance en 83 s. Luego, fuimos interrumpiendo la red desde un extremo para encontrar en qué momento dejaba de funcionar, viendo que

por ejemplo si dejábamos dos *relays* aislados con el corte, la red del lado de los ocho *relays* restantes continúa su avance. Pero si cortamos con un *relay* más se detiene en ambos extremos. Así verificamos que si afectamos más de un 20 % del *stake* activo de la red, la misma no puede continuar su avance y se detiene. Para este caso, la red se recuperó en 52 s.

Habiendo visto este comportamiento, nos interesó conocer si era necesario cortar por completo la conectividad entre cada lado de la red o si alcanzaba con aplicar una demora en el enlace que separa a tres *relays* del extremo. Aplicando *delays* incrementales sobre el mismo vimos que a partir de 6 s la red comienza a fallar y no logra generar bloques por la demora en los mensajes entre ambas partes de la red. Con demoras más grandes se mantiene la detención de la red. Al ser levantada la demora, se restaura la generación de bloques al ritmo original en 28 s segundos. Vimos que el tiempo de recuperación disminuye notablemente cuando se trata de una demora en lugar de una interrupción en el enlace. A su vez, para los valores más bajos de demora vemos como la red se degrada a medida que aumentan, siendo afectada notablemente por más que siga produciendo bloques.

Para incrementar la complejidad de nuestro caso, cambiamos la topología lineal por una circular, conformando un anillo físico entre los *relays*. En este caso, debimos afectar dos enlaces en lugar de uno para poder aislar a tres *relays*. Los resultados fueron similares que en el caso de la red lineal, logrando detener el avance al aplicar una demora de 7 s. El tiempo de recuperación en este caso fue de la mitad del lineal (14 s).

Posteriormente, trabajamos con una topología física donde los *relays* se encontraban en una clique. Al aplicar una demora de 7 s sobre todos los enlaces que partían de tres de los *relays* hacia los demás *relays*, logramos que la red se detuviera tal como sucedió en los casos anteriores. Pero resultó notable la disminución del tiempo de recuperación cuando este se elimina: en tan solo 6 s la red ya estaba produciendo bloques nuevamente.

Con el fin de disminuir la cantidad de segundos de demora que debíamos aplicar para detener la red, agregamos al caso anterior demoras entre los *relays* afectados y los nodos que se conectan a ellos, logrando afectarla con solo 4 s (con un tiempo de recuperación de 12 s). Si bien baja la demora necesaria para detener la red, agregamos nuevos enlaces a afectar. En base a esto, nos planteamos qué sucedería si entonces sólo afectáramos los enlaces entre los nodos y los *relays* que queremos atacar, entendiendo que los nodos son los que toman las decisiones y todo mensaje que llega al mismo luego debe ser contestado (siendo así afectado dos veces por la demora). Aquí observamos que con 7 s de *delay* alcanzó para detener nuevamente a la red, pero afectando solo tres enlaces, lo cual es una mejora significativa en el esfuerzo para mantener un ataque similar. Además, el tiempo de recuperación fue más alto, 88 s, aunque tampoco resulta un valor sumamente grande para este proceso (si fuera más alto podría justificar un ataque corto que provoque que la red deba recuperarse muchas veces).

En base a estos resultados, observamos que `Algorand` tiene buenos tiempos de recuperación ante particiones, aunque en el caso de poder afectar un nodo en particular no resultaría costoso mantenerlo fuera de línea. El hecho de que la red presente nodos con un gran porcentaje del *stake* hace que un ataque a esos nodos pueda ser factible, pero nos faltaría poder encontrarlos (es decir, conocer sus IPs).

Partiendo de ese resultado, nos propusimos plantear un escenario donde logramos controlar un *relay* del sistema y quisimos intentar reconstruir la topología lógica en base a los mensajes de los pares a los que se encuentra conectado, pudiendo así potencialmente ubicar a un nodo que tenga una gran cantidad de *stake*.

Realizamos un análisis de los tipos de mensajes que recibe un *relay* y la frecuencia de ellos, encontrando cuál era el óptimo para relacionar el mensaje a una cuenta en particular, pero que sea recibido con frecuencia para tener un buen muestreo en períodos cortos de tiempo.

En primera instancia, trabajamos con redes donde la distribución del *stake* era homogénea. En base a experimentación pudimos determinar un algoritmo inicial para diferenciar aquellas cuentas conectadas al *relay* que nos envía la información, de los que no lo están, y trabajamos con distintas estrategias en cuanto a cómo muestreamos la red con ventanas de observación, variando su tamaño y cantidad para mejorar los resultados (entendiendo el *trade-off* del resultado versus el esfuerzo de cada caso). Así, logramos predecir con perfecta precisión y *recall* las cuentas conectadas en redes de 30 y 60 nodos. Para el caso de 150 nodos, los resultados también fueron muy positivos, logrando muy buen reconocimiento en algunas de las configuraciones. Pero para la red de 300 nodos, si bien logramos buena precisión en la predicción, el *recall* no fue lo suficientemente bueno.

Si bien el caso del *stake* homogéneo es un caso extremo de descentralización completamente equitativa, en la realidad `Algorand` cuenta con una distribución muy centralizada en cuanto a la cantidad de cuentas que mantienen un porcentaje alto del *stake* activo de la red. Con lo cual, repetimos las experiencias anteriores, pero replicando la distribución del *stake* heterogénea de `Algorand`. Con la estrategia utilizada para el caso homogéneo obtuvimos pésimos resultados en cuanto a la predicción de cuentas conectadas a cada *relay*. Por ese motivo, desarrollamos dos nuevas estrategias para lo mismo donde agregamos peso a los mensajes recibidos según el *stake* de las cuentas, entendiendo que las que tienen mayor cantidad deberían enviar más mensajes. Con la estrategia count no logramos mejoras para ningún caso, pero con la estrategia fibo mejoramos notablemente la predicción, llegando a muy buenos valores de precisión y *recall* para escenarios de 30 y 60 nodos. En el caso de 150 se logró buena precisión, pero con bajo *recall*, mientras que en el escenario de 300 nodos empeoraron estos valores.

Sin embargo, a pesar de obtener malos valores de *recall*, pudimos ver que la precisión en algunos casos era buena: nuestras predicciones eran acertadas, pero nos estábamos perdiendo muchos nodos conectados. Lo cual nos hizo cuestionarnos si estos nodos que estábamos detectando positivamente se correspondían con las cuentas con mayor cantidad de *stake*.

En la última parte del trabajo entonces planteamos escenarios de escala intermedia donde queríamos ver si nuestra estrategia funcionaba para detectar si las cuentas con mayor cantidad de *stake* se encontraban conectadas a alguno de los *relays* vecinos (o no). Los resultados obtenidos fueron muy buenos, prediciendo correctamente esos casos en la gran mayoría de las configuraciones de ventanas. Con lo cual, si quisiéramos encontrar a una cuenta en particular, podríamos saber desde un *relay* si la misma se encuentra conectada a alguno de los vecinos del mismo.

Finalmente, más allá de los resultados obtenidos para los experimentos, en este trabajo se lograron crear múltiples herramientas y *script*s que permiten la virtualización de clientes `Algorand` y la emulación de una red física donde correrlos, tal que permite plantear escenarios de tamaño real y más grandes. Con esto, podría ponerse a prueba distintos elementos de la red y verificar así su escalabilidad para el futuro.

5.2. Trabajo futuro

Con respecto a la primer parte del trabajo, creemos que pueden hacerse pruebas de ataques más sofisticados que alteren la red para probar su adaptabilidad, inyectando mensajes maliciosos o alterando enlaces con otras estrategias. También se podrían hacer pruebas con redes más complejas que intenten replicar escenarios con la distribución de los *relays* de `Algorand` (en vistas de que son públicos y podrían aproximarse las latencias entre ellos).

En cuanto a las predicciones de conectados, hay mucho lugar para mejorar la estrategia de detección. La idea de este trabajo fue dar una estrategia inicial para experimentar, pero es algo que podría trabajarse para mejorar.

También se podría estudiar la posibilidad de afectar otros *relays* del sistema con ataques como los vistos de manera que los nodos dejen de conectarse a ellos, salten a otro *relay* del listado y eventualmente lograr que se conecten al que está infectado. De esa manera se podría conseguir la IP del nodo que queremos intervenir.

Otro enfoque podría ser afectar un *relay* e ir forzando las conexiones a los demás *relays*, cambiándolos luego de un tiempo predeterminado y así obteniendo información de todo el sistema.

Estos puntos, si bien no fueron desarrollados en esta tesis, podrían analizarse con las herramientas desarrolladas para el mismo, abriendo camino a futuros trabajos con `Algorand`.

Apéndice

A. DATOS DE LA RED PRINCIPAL DE ALGORAND (*MAINNET*)

Para este trabajo se tomaron los datos de la red al 6 de noviembre de 2022, realizando una captura de la misma y trabajando con lo observado para ese momento.

A.1. Distribución del *stake*

Se describe la distribución del *stake* en la red oficial de `Algorand` para las principales cuentas de la misma, representando el 99,84 % del *stake* activo en el consenso en ese momento. Con ellos se armó la distribución de *stake* de las redes experimentales con distribución heterogénea de este trabajo.

La cantidad total de *algos* participando del consenso era de 1981637848, se expresan los valores de cada una de las cuentas disponibles, con su porcentaje del total del *stake* y el porcentaje del *stake* representado hasta esa fila de la tabla (incluyendo el valor de la misma).

	Account	**Stake**	**Stake pct**	**Accum *stake* pct**
1	B23XLXSA3MBU3SLP5EF2HE7JR...	99502994	5.02 %	5.02 %
2	SJYNEQ36GVJBQ3YRZTVQGMFQD...	83856812	4.23 %	9.25 %
3	OFB2SMWDDB3V4RYRSC6EXJYB2...	70403976	3.55 %	12.81 %
4	2JBQSMWAZANNM5U2P2DPCBVJ7...	68332792	3.45 %	16.25 %
5	NHHLK67CYONVDXT4H5LNXDHDB...	59343166	2.99 %	19.25 %
6	2A2XRMRP3XR5L7LUP5AGZHHRU...	51793142	2.61 %	21.86 %
7	4L4UCO2NDK2UGGW4MHLMJJYYC...	50965725	2.57 %	24.43 %
8	GBGP7LPZOMDMQHZZIUZR7SXDM...	50965724	2.57 %	27.01 %
9	YEUZBQWVSIYEVXKRGIQRJ25XT...	50965723	2.57 %	29.58 %
10	4UQTDS2M46UT3JL2UDOXFJQFM...	50965722	2.57 %	32.15 %
11	MK2Z5BTVQZE6SWXSUPFVOMKTR...	50965720	2.57 %	34.72 %
12	TEI4MPLR7R4RZZVQOYXC54DNM...	50965718	2.57 %	37.29 %
13	47ONESQNZUNHCUUVE2EC2VZAJ...	50965717	2.57 %	39.87 %
14	CBOQNBM6PCBHVGFQV5OJ4LLIL...	50760939	2.56 %	42.43 %
15	5IIFGXH2CJNRRTZJ4SFATW52X...	47098601	2.38 %	44.80 %
16	4SOZXGYC5MGZ4S24LX3MHOLYV...	46695097	2.36 %	47.16 %
17	F3ORVOGS5JQ5LYIXBUTIEYRSJ...	45844395	2.31 %	49.47 %
18	66UYBBD5LBKCUVMY2I5WWIUFX...	43239943	2.18 %	51.66 %
19	EDFLHVE46UHWZCP5AVB4LT4HA...	40101800	2.02 %	53.68 %
20	EOEMN7CQDFBYZ4N7VLJVW7KYP...	40000003	2.02 %	55.70 %
21	WSRGINBJRGOWSIRAN6N6XSPJR...	39999997	2.02 %	57.72 %
22	W425PRGCPQFAKWSN7D2QKGKBH...	38303749	1.93 %	59.65 %
23	LO4DZTV7XSBJSEVEMFETUENPX...	36375788	1.84 %	61.49 %
24	7N34NWZ4M5E5WU7J2M2VHAAWV...	34967521	1.76 %	63.25 %
25	`Algorand` Inc: 203	34849794	1.76 %	65.01 %
26	YX5KZSZT27L7WZAW7TNONVDZH...	34381795	1.74 %	66.74 %

Tab. A.1: Cantidad de *stake*, porcentaje y porcentaje acumulado para las cuentas de la red `Algorand` para la fecha del 6 de noviembre de 2022. (Parte 1 de 2)

	Account	Stake	Stake pct	Accum *stake* pct
27	4IZRTUO72JY5WH4HKLVDQSKIV...	34132381	1.72 %	68.47 %
28	7LCI345GLLNCNMZJQZ4627GR2...	29999996	1.51 %	69.98 %
29	JXLDZF3DRFKL4E7ZBVWHVKTYM...	29999995	1.51 %	71.49 %
30	AH5NLMPWZPL4DLCK33PH4KZOY...	26993727	1.36 %	72.86 %
31	Algorand Inc: 205	26265234	1.33 %	74.18 %
32	Algorand Inc: 159	26064191	1.32 %	75.50 %
33	Algorand Foundation 23	23818384	1.20 %	76.70 %
34	Algorand Inc: 158	23760747	1.20 %	77.90 %
35	Algorand Inc: 189	21690388	1.09 %	78.99 %
36	IQ5RFRLGS4H6Q5VGEKWAF2LZQ...	19999999	1.01 %	80 %
37	THT2VJC76P7ORURMA25F6RCHH...	19999999	1.01 %	81.01 %
38	Algorand Inc: 210	19849855	1 %	82.01 %
39	Algorand Foundation 20	18084888	0.91 %	82.92 %
40	Algorand Foundation 13	18084888	0.91 %	83.84 %
41	Algorand Foundation 17	18084888	0.91 %	84.75 %
42	Algorand Foundation 19	18084888	0.91 %	85.66 %
43	Algorand Foundation 14	18084888	0.91 %	86.58 %
44	Algorand Foundation 15	18084888	0.91 %	87.49 %
45	Algorand Foundation 21	18084888	0.91 %	88.40 %
46	Algorand Foundation 18	18084888	0.91 %	89.31 %
47	Algorand Foundation 16	18084887	0.91 %	90.23 %
48	I5LZFHEOVMNOJ5TWG7D2RM624...	15000001	0.76 %	90.98 %
49	PLEXAN2222RLHBJE5CTTJAZ2C...	15000000	0.76 %	91.74 %
50	Algorand Foundation 22	12418636	0.63 %	92.37 %
51	R2NC5NEJYFOQ3AIC5QIPAIPZD...	12234926	0.62 %	92.98 %
52	HQQRVWPYAHABKCXNMZRG242Z5...	10473777	0.53 %	93.51 %
53	IQ3SVTCO4MBIFWRDHNWYE6VXF...	10257289	0.52 %	94.03 %
54	7GI4GMAFMI6HP46FJHGXSPTU6...	10256430	0.52 %	94.55 %
55	46U5SGV73FR6DR4VS2LSVPZOT...	10256307	0.52 %	95.07 %
56	WHJRSOUX4P7HQBNGR6ZC3FKS3...	10256287	0.52 %	95.58 %
57	RNSMYL2DB56OXZ7TNOB25SMYO...	10000001	0.50 %	96.09 %
58	EYHLFWJCJOIIAGKVHSGUBLOG2...	10000001	0.50 %	96.59 %
59	IOUSAKUXEFKWZE7QON3AJYNEG...	10000000	0.50 %	97.10 %
60	37XETSSJAH3IXUDJ6RJBID3UJ...	9999999	0.50 %	97.60 %
61	YA73DAAA6VH4ODDNXUPBCHKRG...	9669729	0.49 %	98.09 %
62	Algorand Inc: 224	6186201	0.31 %	98.40 %
63	UUOB7ZC2IEE4A7JO4WY4TXKXW...	5509247	0.28 %	98.68 %
64	6F5GHHGDIPZWY5DNBWSVA75AL...	4976251	0.25 %	98.93 %
65	7O6CTSDNCVWQACJYYQRESHMK2...	4489585	0.23 %	99.16 %
66	Algorand Inc: 223	4206876	0.21 %	99.37 %
67	MVJD3XLUA6PWGXXXRMPGUZAKQ...	3002028	0.15 %	99.52 %
68	OCSSD3JEBBBETMXOJN2YS2S3K...	2500004	0.13 %	99.65 %
69	4ZODDDUGJIOXZF6GMV42OG3I3...	1499587	0.08 %	99.72 %
70	CJS3CGUPR5G2WM7I35YDLN46O...	1034689	0.05 %	99.77 %
71	Algorand Inc: 126	604193	0.03 %	99.81 %
72	XO2RXKXBHWBAVX43TAI5PKLKS...	412676	0.02 %	99.83 %
73	MBUTXAIOCTMKMZVKVCOWMLU3M...	360019	0.02 %	99.84 %

Tab. A.2: Cantidad de *stake*, porcentaje y porcentaje acumulado para las cuentas de la red Algorand para la fecha del 6 de noviembre de 2022. (Parte 2 de 2)

Bibliografía

[1] Bitcoin hashrate historical chart. `https://bitinfocharts.com/comparison/bitcoin-hashrate.html`. Accessed: 8 de abril de 2024.

[2] The real-world costs of the digital race for bitcoin. `https://www.nytimes.com/2023/04/09/business/bitcoin-mining-electricity-pollution.html`. Accessed: 8 de abril de 2024.

[3] Cambridge bitcoin electricity consumption index. `https://ccaf.io/cbnsi/cbeci`. Accessed: 8 de abril de 2024.

[4] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120 – 126, 1978. Cited by: 11392; All Open Access, Green Open Access.

[5] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):243 – 264, 1987. Cited by: 3456; All Open Access, Bronze Open Access.

[6] Victor S. Miller. Use of elliptic curves in cryptography. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 218 LNCS:417 – 426, 1986. Cited by: 2812; All Open Access, Bronze Open Access.

[7] Mauro Conti, Ankit Gangwal, and Michele Todero. Blockchain trilemma solver algorand has dilemma over undecidable messages. pages 1–8, 08 2019.

[8] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, pages 382–401, July 1982.

[9] Vicent Sus. Proof-of-stake is a defective mechanism. *SSRN Electronic Journal*, 03 2022.

[10] Silvio Micali. ALGORAND: the efficient and democratic ledger. *CoRR*, abs/1607.01341, 2016.

[11] How does algorand solve the blockchain trilemma? - an explanation by silvio micali on the lex fridman podcast. `https://www.algorand.com/resources/blog/silvio-micali-lex-fridman-algorand-and-the-blockchain-trilemma/`. Accessed: 8 de abril de 2024.

[12] Sharon Goldberg, Leonid Reyzin, Dimitrios Papadopoulos, and Jan Včelák. Verifiable Random Functions (VRFs). Internet-Draft draft-irtf-cfrg-vrf-03, Internet Engineering Task Force, 2018.

[13] Vrf repository. `https://github.com/algorand/vrf`. Accessed: 8 de abril de 2024.

[14] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 51–68, New York, NY, USA, 2017. Association for Computing Machinery.

[15] Jing Chen, Sergey Gorbunov, Silvio Micali, and Georgios Vlachos. Algorand agreement: Super fast and partition resilient byzantine agreement. Technical report, Cryptology ePrint Archive, 2018. Report 2018/377.

[16] Algorand developer docs. `https://developer.algorand.org/docs`. Accessed: 8 de abril de 2024.

[17] Algoexplorer: top accounts. `https://algoexplorer.io/top-accounts`. Accessed: 8 de abril de 2024.

[18] Christian Decker and Roger Wattenhofer. Information propagation in the bitcoin network. In *Proceedings of the IEEE P2P 2013*, pages 1–10. Institute of Electrical and Electronics Engineers, Inc, September 2013.

[19] Joan Antoni Donet, Cristina Pérez-Sola, and Jordi Herrera-Joancomartí. The bitcoin P2P network. In *Financial Cryptography and Data Security*, Lecture Notes in Computer Science, pages 87–102, Berlin, Heidelberg, 2014. Springer.

[20] Andrew Miller and Rob Jansen. Shadow-bitcoin: Scalable simulation via direct execution of multi-threaded applications. In *Proceedings of the 8th USENIX Conference on Cyber Security Experimentation and Test*, CSET'15, pages 7–7, Berkeley, CA, USA, 2015. USENIX Association.

[21] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, and Emin Gün. On scaling decentralized blockchains. In *Financial Cryptography and Data Security*, volume 9604 of *Lecture Notes in Computer Science*, pages 106–125, Berlin, Heidelberg, August 2016. Springer.

[22] Yonatan Sompolinsky and Aviv Zohar. Accelerating bitcoin's transaction processing. fast money grows on trees, not chains. *IACR Cryptology ePrint Archive*, 2013:881, 2013.

[23] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques*, volume 9057 of *Lecture Notes in Computer Science*, pages 281–310, Berlin, Heidelberg, 2015. Springer.

[24] Aggelos Kiayias and Giorgos Panagiotakos. Speed-security tradeoffs in blockchain protocols. Technical report, Cryptology ePrint Archive, 2015. Paper 2015/1019.

[25] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol with chains of variable difficulty. Cryptology ePrint Archive, Report 2016/1048, 2016. `https://eprint.iacr.org/2016/1048`.

[26] Marco Vanotti. Un avance hacia entornos de gran escala para experimentos con criptomonedas. Master's thesis, Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, 2016.

[27] Silvio Vileriño. Estudio de los límites de generación de bloques en blockchain. Master's thesis, Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, 2017.

[28] Kai Otsuki, Yusuke Aoki, Ryohei Banno, and Kazuyuki Shudo. Effects of a simple relay network on the bitcoin network. In *Proceedings of the Asian Internet EngineeringConference*, pages 41 – 46. Association for Computing Machinery, 2019.

[29] Nicolás De Carli. Sobre los límites del tiempo entre bloques en bitcoin. Master's thesis, Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, 2019.

[30] Julio Augusto Mascitti. Impacto de la red fibre en bitcoin: ¿los mineros crearon un sistema casi centralizado? Master's thesis, Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, 2022.

[31] Andrew Lewis-Pye and Tim Roughgarden. Resource pools and the CAP theorem. *CoRR*, abs/2006.10698, 2020.

[32] Cristian Lepore, Michela Ceria, Andrea Visconti, Udai Pratap Rao, Kaushal Shah, and Luca Zanolini. A survey on blockchain consensus with a performance comparison of pow, pos and pure pos. *Mathematics*, 8:1782, October 2020.

[33] Jiseong Noh, Donghwan Kwon, Soohwan Cho, and Neo C. K. Yiu. Network participation and accessibility of proof-of-stake (pos) blockchains: A cross-platform comparative analysis. 2023.

[34] Francesco Mogavero, Ivan Visconti, Andrea Vitaletti, and Marco Zecchini. The blockchain quadrilemma: When also computational effectiveness matters. pages 1–6, 09 2021.

[35] Nicola Dimitri. Proof-of-stake in algorand. *Distributed Ledger Technologies: Research and Practice*, 1, 12 2022.

[36] Nicola Dimitri. The economics of consensus in algorand. *FinTech*, 1(2):164–179, 2022.

[37] Mayank Pandey, Rachit Agarwal, Sandeep Kumar Shukla, and Nishchal Kumar Verma. Reputation-based pos for the restriction of illicit activities on blockchain: Algorand usecase. *CoRR*, abs/2112.11024, 2021.

[38] Aline Gouget, Jacques Patarin, and Ambre Toulemonde. Unpredictability properties in algorand consensus protocol. pages 1–3, 05 2021.

[39] Maryam Abbasihafshejani, Mohammad Hossein Manshaei, Mohammad Rahman, Kemal Akkaya, and Murtuza Jadliwala. On algorand transaction fees: Challenges and mechanism design. 05 2022.

[40] Mehdi Fooladgar, Mohammad Hossein Manshaei, Murtuza Jadliwala, and Mohammad Ashiqur Rahman. On incentive compatible role-based reward distribution in algorand. *CoRR*, abs/1911.03356, 2019.

[41] Caspar Schwarz-Schilling, Joachim Neu, Barnabé Monnot, Aditya Asgaonkar, Ertem Tas, and David Tse. *Three Attacks on Proof-of-Stake Ethereum*, pages 560–576. 10 2022.

[42] Yongge Wang. Another look at ALGORAND. *CoRR*, abs/1905.04463, 2019.

[43] Suhyeon Lee and Seungjoo Kim. Shorting attack: Predatory, destructive short selling on proof-of-stake cryptocurrencies. *Concurrency and Computation: Practice and Experience*, 35, 09 2021.

[44] F. Harary and G. Gupta. Dynamic graph models. *Mathematical and Computer Modelling*, 25:79–87, 04 1997.

[45] Camil Demetrescu, David Eppstein, Zvi Galil, and Giuseppe F. Italiano. *Dynamic Graph Algorithms*, page 9. Chapman & Hall/CRC, 2 edition, 2010.

[46] Algorand code repository for version 3.0.1. `https://github.com/algorand/go-algorand/releases/tag/v3.0.1-stable`. Accessed: 8 de abril de 2024.

[47] Maximiliano Geier and Esteban Mocskos. Sherlockfog: Finding opportunities for MPI applications in fog and edge computing. In Esteban Mocskos and Sergio Nesmachnow, editors, *High Performance Computing*, pages 185–199, Cham, 2018. Springer International Publishing.