FACULTAD DE INGENIERIA
Universidad de Buenos Aires

TALLER DE PROGRAMACION III (7561)

Trabajo Práctico:
InterPlanetary File System (IPFS)

Docentes:

- Profesor Adjunto: Manuel Camejo
- Jefe de Trabajos Prácticos: Leandro Ferrigno

Alumno:

- Putaro Rafael Alejandro - 84236

Fecha: Noviembre de 2025

Índice:

1. Introducción:

El crecimiento exponencial del tráfico de datos y la necesidad de sistemas más eficientes, resilientes y descentralizados han impulsado el desarrollo de nuevas arquitecturas para el almacenamiento y distribución de información. Frente a los modelos centralizados tradicionales, que presentan cuellos de botella, dependencia de servidores únicos y vulnerabilidades ante fallos o censura, surgen alternativas basadas en redes punto a punto (P2P). En este contexto, IPFS (InterPlanetary File System) se posiciona como un protocolo innovador que permite almacenar y compartir datos de forma distribuida, utilizando identificadores de contenido (CIDs) en lugar de ubicaciones de servidores.

Uno de los pilares fundamentales del funcionamiento de IPFS es su sistema de enrutamiento de contenido, que depende de una DHT (Distributed Hash Table) basada en el protocolo Kademlia. Este protocolo permite localizar nodos que almacenan datos específicos de manera eficiente, escalable y robusta ante la dinámica constante de entrada y salida de nodos en la red. Kademlia utiliza una métrica de distancia XOR para organizar los nodos y optimizar las búsquedas, minimizando el número de saltos necesarios para encontrar un recurso.

El objetivo de este trabajo práctico es analizar y comprender cómo Kademlia facilita la operación de IPFS, explorando su implementación, ventajas y desafíos en redes descentralizadas. A través de una aproximación teórico-práctica, se busca demostrar la eficacia del protocolo en la resolución de rutas de contenido y su contribución a la descentralización efectiva de la web.

2. Desarrollo:

Fundamentos de IPFS (InterPlanetary File System): IPFS es un protocolo de red descentralizado diseñado para hacer la web más rápida, segura y resistente a la censura. Su propósito principal es reemplazar o complementar el modelo cliente-servidor tradicional, donde los archivos se alojan en servidores centralizados, por un sistema basado en contenido (content-addressed), en el que cada archivo tiene un identificador único (CID) basado en su contenido.

En lugar de buscar archivos por su ubicación (como una URL), en IPFS se busca por su huella digital (hash). Esto significa que si dos archivos son idénticos, tendrán el mismo CID y solo se necesita dicho identificador para encontrarlos en cualquier nodo de la red que los esté compartiendo.

La red descentralizada de IPFS funciona mediante un sistema peer-to-peer (P2P), donde cada participante (nodo o par) puede almacenar, servir y solicitar archivos. Cuando un nodo quiere encontrar un archivo, utiliza una DHT (Distributed Hash Table) basada en el protocolo Kademlia para localizar qué nodos cercanos en la red tienen el contenido solicitado. Esta DHT permite búsquedas eficientes con pocos saltos, incluso en redes con miles de nodos.

Además, IPFS permite la distribución eficiente de datos, ya que los archivos se dividen en bloques pequeños que pueden descargarse desde múltiples nodos simultáneamente.

Algunos casos de uso de IPFS:

- Alojamiento de sitios web y contenido estático.
- Distribución de libros y bibliotecas digitales: Library Genesis permite compartir documentos científicos y fue utilizada recientemente por Meta para entrenar a su IA.
- Plataformas de música descentralizadas: Audius.
- Proyectos de identidad digital: Microsoft ION.
- Tiendas descentralizadas: Brave utiliza IPFS y el protocolo Origin para alojar su tienda de productos descentralizada, y desde 2021, su navegador incluye soporte nativo para IPFS, permitiendo a los usuarios acceder directamente a contenidos IPFS.
- Redes de almacenamiento cooperativo: Filecoin.
- Navegación móvil: Opera para Android incluye compatibilidad por defecto con IPFS.
- Proyectos anticensura: Durante el referéndum independentista catalán en 2017, el Partido Pirata Catalán replicó su sitio web en IPFS para eludir la orden de bloqueo del Tribunal Superior de Justicia de Catalunya.

Fundamentos de Kademlia: Kademlia es un protocolo de tabla hash distribuida (DHT) diseñado para redes P2P descentralizadas, creado en 2002 por Petar Maymounkov y David Maziéres. Su principal objetivo es permitir la localización eficiente de nodos y recursos en una red grande y dinámica, sin depender de servidores centrales.

Cada nodo en Kademlia tiene un ID único de 160 bits, generado aleatoriamente. La "distancia" entre dos nodos se calcula mediante la operación XOR entre sus IDs, lo que permite una métrica simétrica y eficiente para organizar la red.

El protocolo se basa en tablas de enrutamiento (k-buckets), donde cada nodo almacena información de otros nodos agrupados por cercanía (distancia XOR). Estas tablas permiten búsquedas rápidas en un número logarítmico de pasos, consultando solo los nodos más cercanos al destino en cada iteración. Por ejemplo en una red con 10 millones de nodos una búsqueda puede ser resuelta con menos de 20 saltos en la red de nodos.

Kademlia utiliza cuatro mensajes principales:

- **PING:** Verifica si un nodo está activo.
- **STORE:** Pide a un nodo que respalde un par clave-valor.
- **FIND NODE:** Busca los nodos más cercanos a una clave dada.
- **FIND VALUE:** Similar a FIND NODE, pero si el nodo tiene el valor asociado a la clave, lo devuelve directamente.

Cuando se busca un recurso, el nodo realiza una búsqueda iterativa: consulta a los nodos más cercanos conocidos, estos devuelven otros más cercanos, y así sucesivamente hasta encontrar el destino. Este mecanismo es escalable, tolerante a fallos y resistente a ataques, ya que no hay puntos únicos de falla.

Además, Kademlia soporta almacenamiento replicado y caché de contenido popular, mejorando la disponibilidad y reduciendo la carga en nodos específicos.

3. Implementación práctica:

Para observar el comportamiento del protocolo IPFS en funcionamiento se optó por una implementación limitada a unas pocas decenas de nodos con un número de clientes concurrentes reducido los operan sobre dicha red subiendo y descargando un conjunto de archivos.

Dicha implementación está dada por una serie de contenedores de Docker, desarrollados fundamentalmente en Golang, los cuales son análogos de pares en una red IPFS que interactúan entre sí mediante llamadas a procedimiento remoto (en este caso gRPC).

Por otro lado los clientes concurrentes que consumen los servicios de los nodos (o pares) de esta red IPFS interactúan con los mismos alternando entre llamados gRPC y la utilización de socket's TCP. Estos clientes también están dados por contenedores Docker desarrollados en Golang.

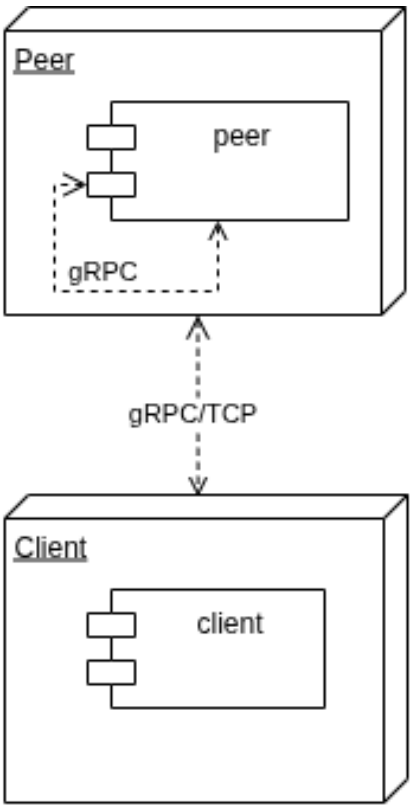


Diagrama de despligue: Se puede observar como los pares interactúan entre sí mediante gRPC y con los clientes a través de gRPC y TCP.

Para la recolección de métricas se utilizó tanto Prometheus como cAdvisor, mientras que para su visualización se ha optado por Grafana.

El sistema consiste principalmente en un paquete “peer” que contiene la implementación del IPFS y un paquete de “client” que consume los servicios de la red de nodos. Ambos paquetes se apoyan en el módulo “common” el cuál contiene módulos dedicados a la comunicación, la gestión de archivos, inicialización de métricas, la implementación de contactos (datos de cada nodo como la url y su identificación), gestión de claves, un planificador de tareas y las operaciones gRPC necesarias tanto para subir como para descargar archivos.

Adicionalmente se tiene un paquete “protobuf” que define los servicios gRPC, los resultados, operandos e implementa una serie de funciones que facilitan el uso de este protocolo en la presente implementación.

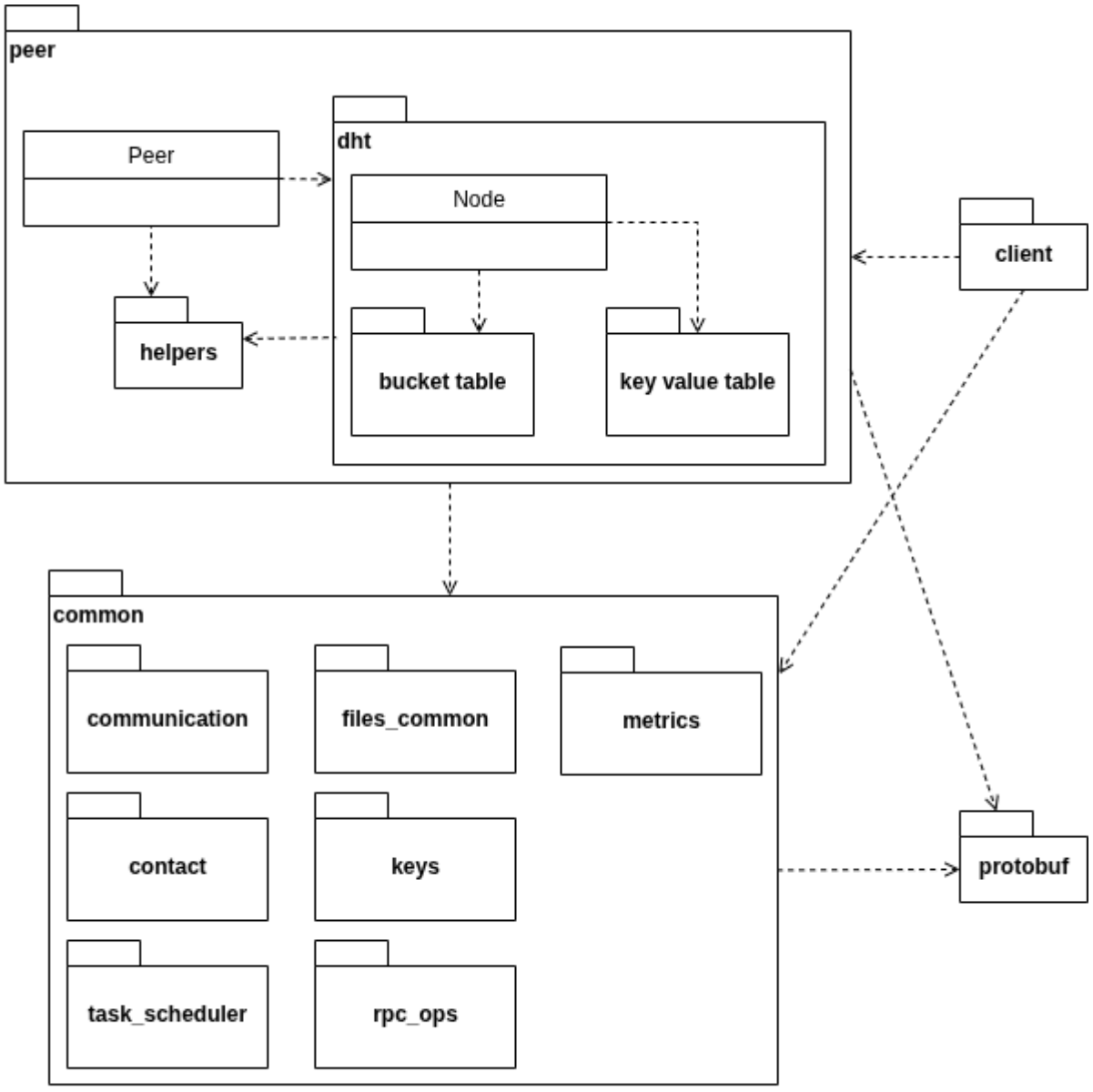


Diagrama de Paquetes

Descripción general de los pares:

Cada par o “peer” es uno de los tantos nodos que conforman la red del IPFS. La ejecución de cada una de estas unidades está dada principalmente por tres hilos ligeros de Golang (goroutines) bajo las siguientes responsabilidades:

- Recibir archivos vía TCP.
- Escuchar peticiones gRPC.
- Desencolar tareas programadas y proceder a su ejecución.

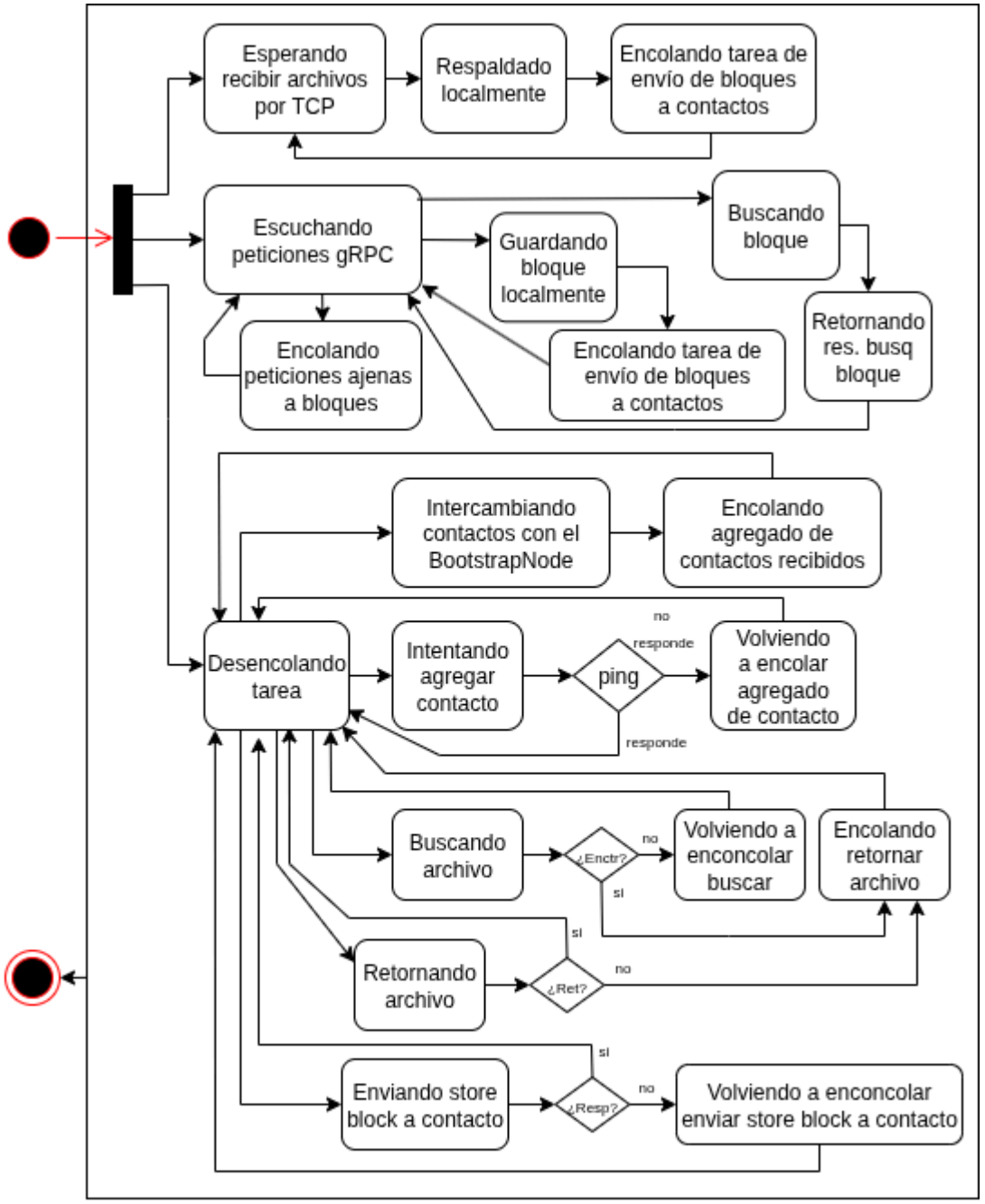


Diagrama de estados de un "peer"

En primer lugar se debe destacar que los pares fueron implementados buscando garantizar la ausencia de starvation, ya que la ejecución del protocolo Kademlia implica que un nodo podría pasar mucho tiempo bloqueado para responder a un mensaje, por ejemplo el envío de las partes de un archivo a todos sus contactos, la búsqueda y descarga de un archivo asociado a una clave específica o bien esperando la respuesta de un mensaje de ping a un contacto específico. Para lograr este objetivo se decidió que la ejecución efectiva de aquellas funcionalidades que puedan ser diferidas o demoradas se encole como una tarea programada. De esta manera cuando se recibe una solicitud gRPC algunas se encolan difiriendo su ejecución y otras implican que el nodo se bloquee hasta completarla.

Cuando una tarea programada es desencolada por el hilo que ejecuta el planificador (task_scheduler) puede derivar en que su ejecución sea fallida (por ejemplo al encontrarse un nodo objetivo de un ping ocupado) implicando que la misma requiera un reintento posterior, por lo cuál la misma vuelve a ser encolada. Estos reintentos son limitados, y permiten alternar entre tareas que pueden ser ejecutadas de manera exitosa en un momento dado y aquellas que no.

En un contexto de congestión la ejecución del protocolo Kademlia podría sufrir la presencia de deadlocks, pero como en la implementación de las operaciones rpc se incluyeron timeouts los mismos no son posibles.

Como se puede observar en el diagrama de estados para un “peer”, de la página anterior, la vida de un nodo podría iniciar (de hecho lo hace) con una tarea de compartir contactos con su “bootstrap node” (nodo que proporciona información inicial a todo aquél nodo que se incorpore a la red) precargada la cuál será desencolada y ejecutada. En una ejecución exitosa esto implica que el nodo tendrá a su disposición una lista de contactos a ser agregados, lo cuál implica para cada uno ellos la creación de una tarea encargada de hacer el ping y agregar efectivamente el contacto a la bucket-table.

Otro detalle a destacar es que se programó el sistema para que acepte que en el archivo de configuración se indique que los nodos sean agrupados para que dependan de un mismo “bootstrap node” secundario. Por lo tanto se tiene una estructura jerárquica de dos niveles para la operación de compartir contactos, la cual permite reducir la carga sobre el “bootstrap node” principal.

Arranque del sistema y construcción de las bucket-table’s:

Durante el arranque de un “peer” se presenta una mayor frecuencia de operaciones de compartir contactos con su “bootstrap node” para luego disminuir la misma, esto permite garantizar que la red converja rápidamente en este modelo de prueba.

Si observamos la figura 3.2 (40 pares agrupados de a 10), de la página posterior, podemos observar que para 40 pares la principal ventaja de la agrupación por “bootstrap-node” radica en que los pares consiguen incorporar contactos al mismo

ritmo, mientras que en la figura 3.1 (40 pares sin agrupación) podemos observar que ante la saturación del bootstrap node algunos pares extienden el tiempo para incorporar la misma cantidad de contactos. El tiempo en el que alcanza una cantidad de contactos estables se en ambos caso cerca de los 4 minutos de iniciado el sistema.

Fig 3.1: 40 pares sin agrupar

Fig 3.2: 40 pares agrupados de a 10

Este comportamiento se nota aún más cuando se sube la cantidad de pares a 50 como podemos observar en las figuras 3.3 y 3.4.

En ambos casos se alcanza una cantidad de contactos estables aproximadamente a los 13 minutos de iniciado el sistema.

De esta manera el sistema sin agrupamiento, con 50 pares, se satura rápidamente al intentar compartir contactos quedando algunos nodos relegados, mientras que bajo agrupamiento se consigue una evolución más uniforme. Esto se debe a que proliferan muchos mensajes entre el bootstrap único y el resto de nodos.

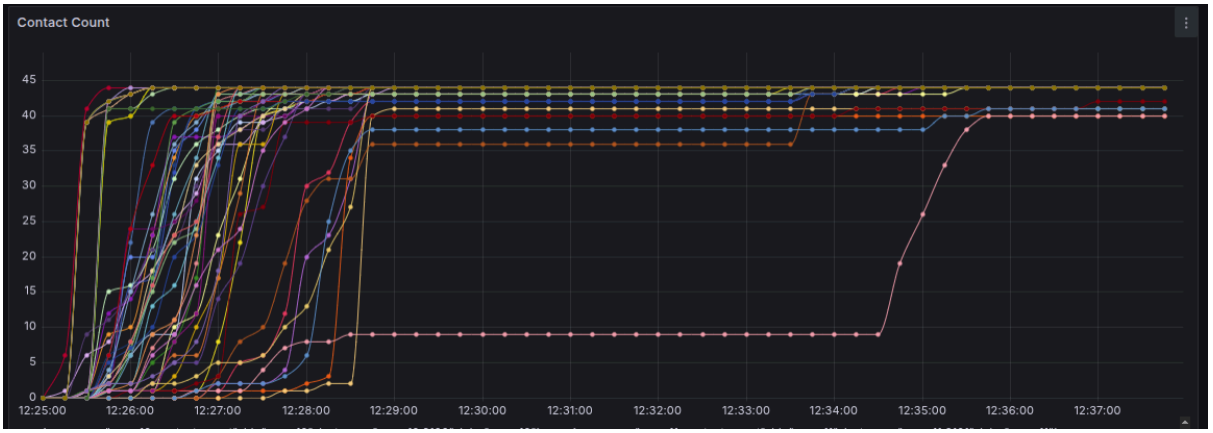


Fig 3.3: 50 pares sin agrupar

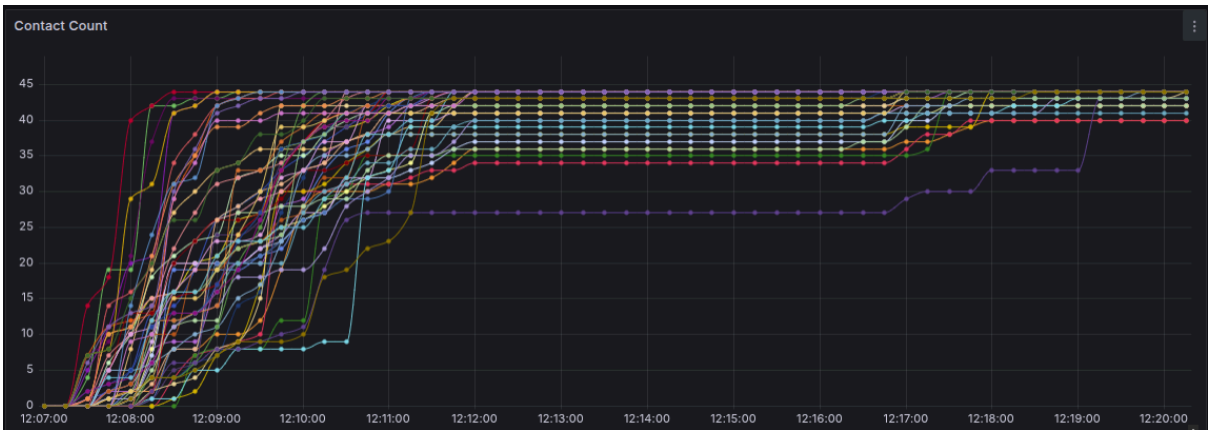


Fig 3.4: 50 pares agrupados de a 10

Se observa que la utilización de nodos secundarios puede ocasionar que las “bucket table’s” lleguen a una cantidad estable de contactos ligeramente menor al caso sin agrupamiento. De todos modos la intención de un IPFS no es que todos los nodos se conozcan, sino disponer en la bucket-table de los contactos activos que permitan construir un camino, a través de la red, hasta el contenido asociado a cualquier clave que los clientes soliciten, por lo tanto no sólo se trata de cantidad sino además de la calidad de los mismos. De hecho como Kademlia consiste de una tabla de hash distribuída la calidad de la tabla de cada individuo aporta a la eficacia y eficiencia de toda la red, por eso es preferible un comportamiento uniforme de los nodos al momento de construir sus “bucket-table’s”.

Carga de archivos a la red:

Los clientes concurrentes a través del protocolo gRPC solicitan a cualquiera de los pares la subida de un archivo con su nombre. El par implicado retorna al cliente la clave asociada al archivo y una URL a la cual por medio del protocolo TCP se debe transferir el archivo.

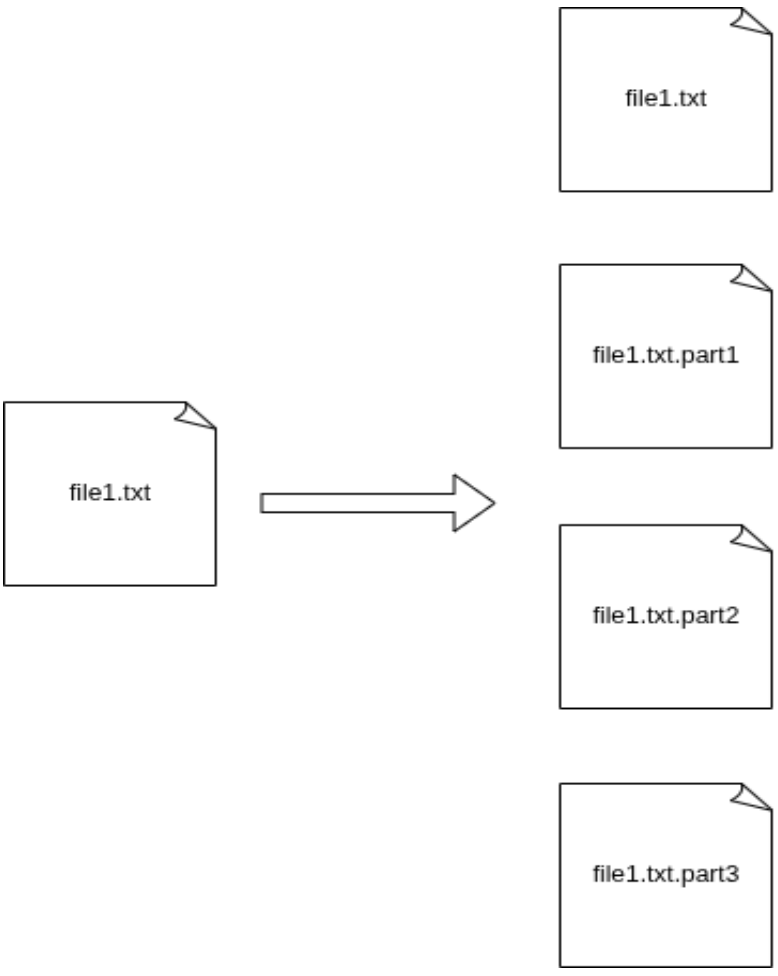
Como se dijo, en la descripción general de los pares, cada uno de los nodos tiene un hilo encargado de escuchar la llegada de archivos por TCP. Este hilo se encarga

de respaldar el archivo localmente y generar una tarea programada encargada de fragmentar dicho archivo y subirlo a la red de nodos.

En este contexto los fragmentos son llamados bloques cada uno los cuales están estructurados de la siguiente manera:

Block Key
Next Block Key
Payload

El “payload” de cada uno de estos bloques está dado por 256 KBytes del archivo original. De esta manera un archivo subido a la red de nodos está dado por un cierto número de bloques enlazados por clave de bloque. El primer bloque siempre mantiene el nombre del archivo original (quedando asociado al nombre del archivo original), el resto agregan la extensión “.part<n>” y el último tiene una “next block key” nula. De esta manera si subo un archivo llamado “file1.txt” con un tamaño de 1 Mb el mismo será descompuesto en 4 archivos con un payload de 256 Kb llamados: “file1.txt”, “file1.txt.part1”, “file1.txt.part2” y “file1.txt.part3”.



En el siguiente diagrama de secuencia se puede observar como un archivo subido por el cliente a un nodo dado implica que se encole una tarea de agregado de archivo. Esta tarea será posteriormente desencolada para ser ejecutada bloqueando el hilo implicado hasta finalizar el envío de los bloques generados a todos los contactos correspondientes.

Diagrama de secuencia subida de archivo a la red de nodos

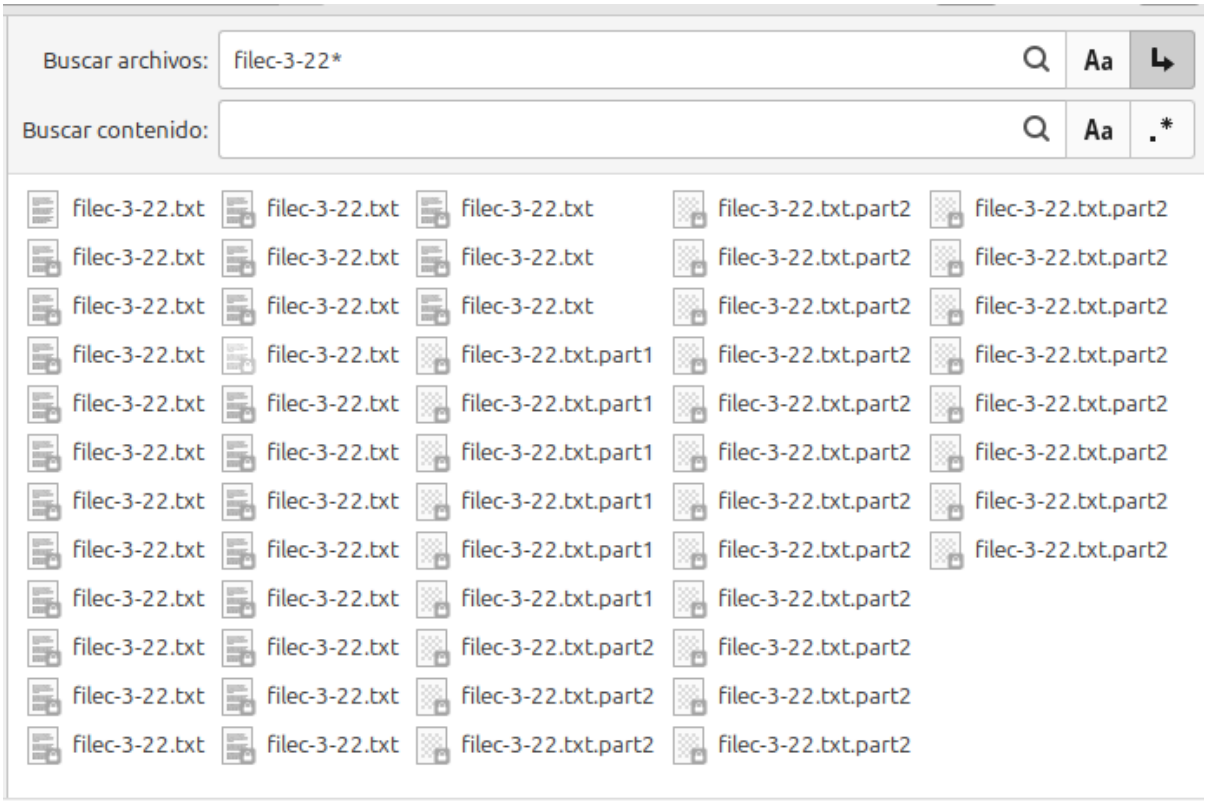
En caso de que no se logre contactar a ninguno de los contactos seleccionados el bloque se almacena localmente y se agrega a la tabla clave-valor.

Cabe destacar que esta es una de las tareas que más tiempo deja bloqueado el hilo de ejecución asociado al task-scheduler. De hecho si se hubiese implementado sin tareas programadas podría haber bloqueado toda una subred de nodos, ya que Kademlia nos indica que se deben llamar los nodos en cascada según la bucket-table.

Descarga de archivos:

En Kademlia la descarga de archivos se realiza utilizando una key asociada al archivo la cuál es provista por el nodo al cuál fue subido. En la sección anterior se explicó que subir un archivo implica la propagación de sus partes (bloques) a través de toda la red. De esta manera la descarga se verá afectada por la dinámica de distribución (en tiempo y amplitud) de los bloques a la red. Es más en Kademlia la operación “store” puede implicar la replicación de bloques aumentando la disponibilidad de los mismos y por lo tanto bajando los tiempos de descarga.

En el presente sistema se agruparon en la carpeta“tmp/tp” todos los volúmenes asociados a clientes y pares. De esta manera, a modo de ejemplo, con una búsqueda en dicha carpeta con “el gestor de archivos del sistema operativo” podemos observar que con 40 pares se ha logrado el siguiente nivel de replicación para el archivo “filec-3-22.txt” y sus partes luego de haber sido subido a la red de nodos:



Como se puede observar en el siguiente diagrama de actividades la descarga de un archivo inicia con un llamado a procedimiento remoto desde el cliente a un par determinado. El llamado es atendido por un hilo dedicado, dentro del par, el cuál encola esta tarea para luego retornar un mensaje de “accept” al cliente.

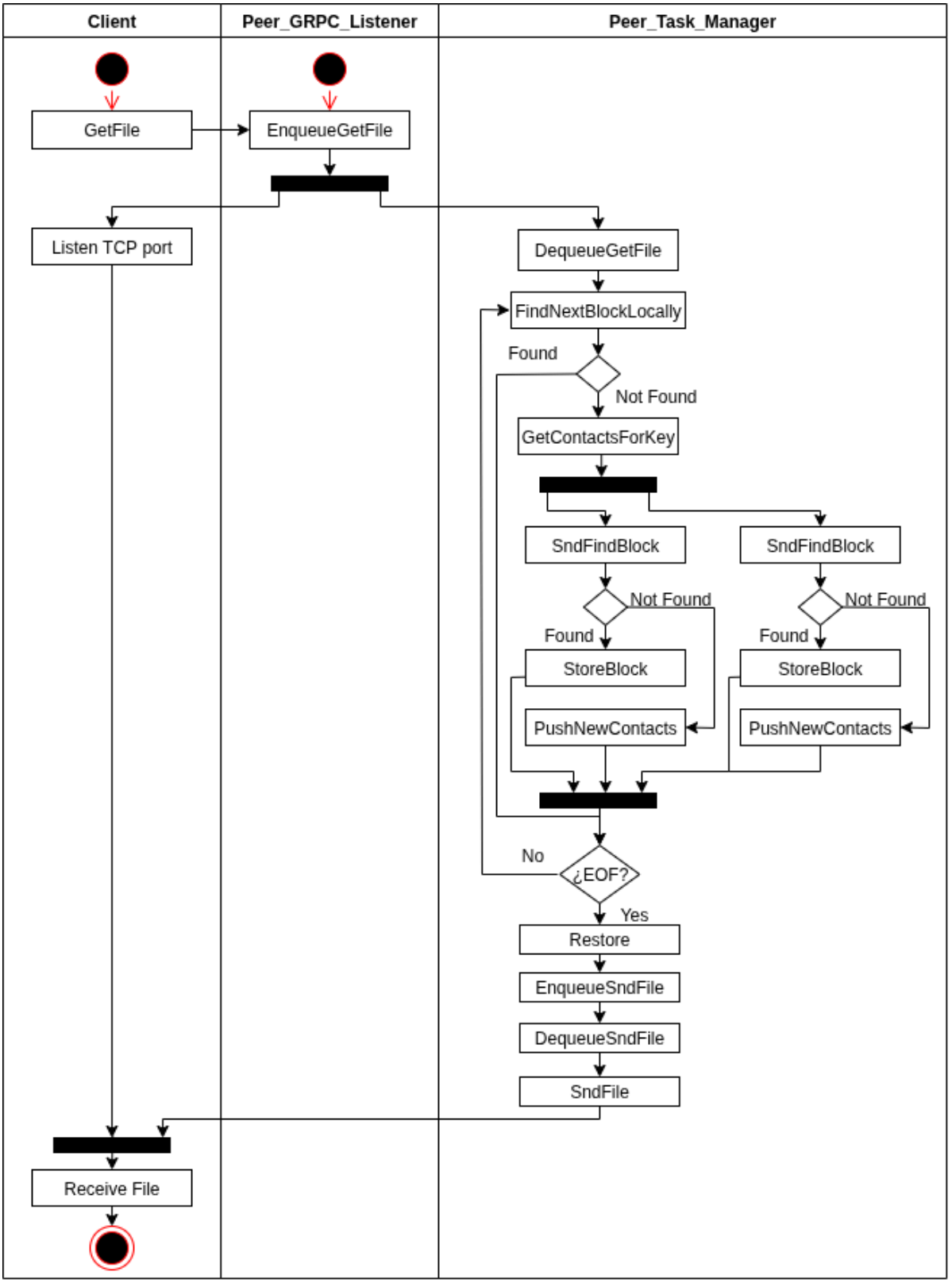


Diagrama de actividades “descarga de archivos”.

El cliente dispone de un hilo dedicado a atender conexiones TCP, siendo la vía establecida para recibir el archivo solicitado. Dentro de cada par existe un hilo encargado de ejecutar el código del “task_manager” el cuál desencolará en algún momento la tarea asociado a la obtención del archivo dado, iniciando la recuperación de los diferentes bloques asociados al mismo. Como se mencionó en la sección anterior el primer bloque tiene asociada la “key” del archivo original y cada uno de los bloques contienen la “key” del siguiente bloque. De esta manera el algoritmo comienza con la búsqueda local del bloque (en la key-value-table) y en caso de encontrarlo se continúa con la búsqueda del siguiente bloque. Si no se encuentra la clave localmente se arma una lista de contactos más cercanos en base al contenido de la “bucket-table”. Esta lista de contactos alimenta a un batería de hilos que se lanzan durante la búsqueda para llamar concurrente a varios contactos con el objetivo de acelerar la búsqueda de bloques. Si alguno de los contactos llamados retorna efectivamente el bloque, el hilo implicado guarda en la carpeta “down” el mismo, mientras que una búsqueda fallida implica que se retornen una lista de contactos (como indica Kademlia) los cuales son agregados a la lista obtenida originalmente desde la “bucket-table” local. En el diagrama no se muestra, pero si en algún momento la lista queda vacía y no se ha obtenido el bloque, está ejecución de la tarea “GetFile” es catalogada como fallida y es sometida a políticas de reintento. Como se muestra en el diagrama una vez obtenido un bloque se repite el ciclo para buscar el siguiente, finalizando una vez que se han obtenido todos los bloques del archivo. El paso posterior está dado por la recuperación del archivo original en base a los bloques descargados, luego de lo cuál se encola una tarea de envío de archivo al cliente que hizo la solicitud gRPC que inició todo el proceso. Cuando esta última tarea es desencolada se envía el archivo mediante un socket TCP al cliente finalizando todo el proceso de descarga de un archivo asociado a una clave dada.

Como se puede concluir en base a lo mencionado anteriormente la recuperación de un archivo en una red IPFS a partir de su clave es un proceso complejo en el cuál participan varios hilos que operan concurrentemente sobre múltiples recursos del nodo haciendo llamados a otros nodos de la red que permanecen un tiempo considerable dedicados exclusivamente a procesar esta solicitud. De esta manera para evitar una saturación de la red se prefirió utilizar tareas programadas y políticas de reintento (administradas por el propio par) en detrimento de que cada una de las búsquedas sea exitosa. Esto implica que se realizan un número limitado de reintentos de llamado a otros nodos de la lista de contactos antes de considerarlos como “no responsivos” en una búsqueda de bloques.

Con el objetivo de observar el comportamiento del sistema durante la búsqueda de archivos se planteó un escenario que está dado por tres clientes concurrentes los cuales suben y descargan 10 archivos cada uno a una red IPFS de 40 nodos agrupados de a 10 nodos por “booostrap-node” secundario.

La generación automática de archivos de entrada ha sido programada para que se pueda extraer de los nombre de los mismos una identificación del 1 al 30, lo que permitirá discriminar a los mismos en las figuras posteriores de la presente sección. Estos archivos están dados por texto, de longitud y contenido aleatorio, generado con un script de Python que se apoya en la librería “lorem-text”.

Como se puede observar en la “figura 3.5” el tiempo descarga promedio ha sido de 7.29 segundos una vez descargados los 30 archivos.

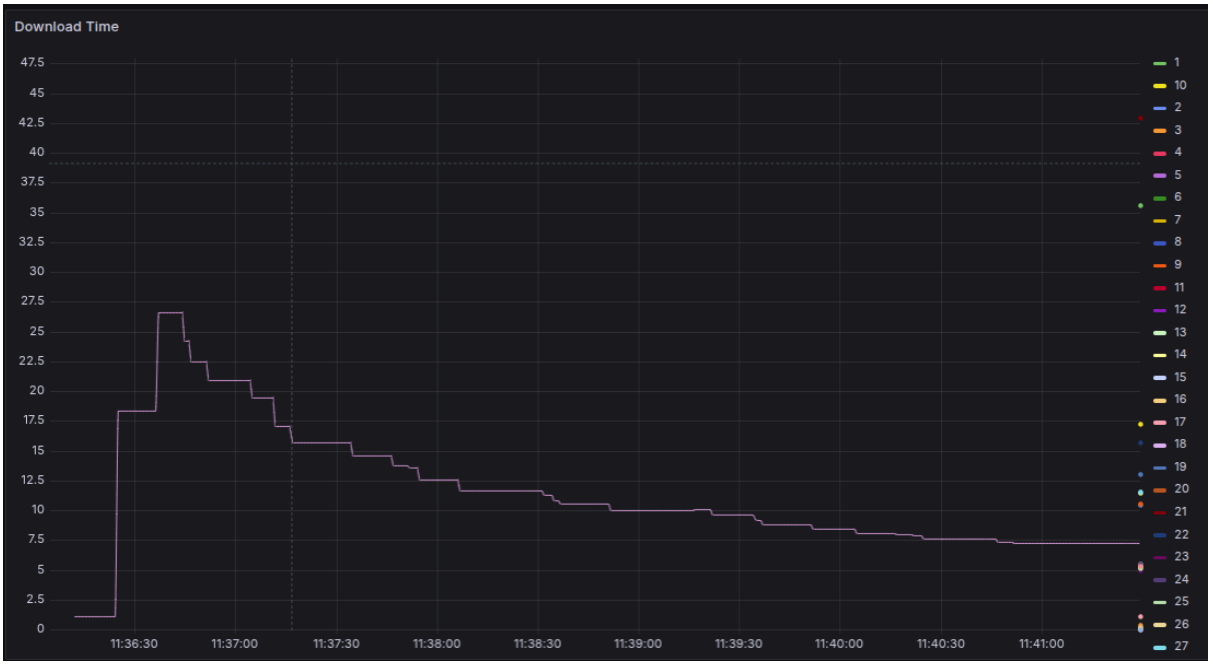


Fig 3.5: Tiempo de descarga 3 clientes, 10 archivos y 40 pares agrupados de a 10

La mayoría de los archivos arrojaron tiempos de descarga por debajo de los 17 segundos, mientras que sólo dos han superado los 30 segundos (los archivos con id 1 y 21).

En la “figura 3.6” se observa para cada archivo los pares que fueron retornando un bloque perteneciente al mismo a lo largo del tiempo. De esta manera podemos inferir que un archivo que presenta actividad en el gráfico durante un amplio rango de tiempo ha tardado más en ser recuperado luego de ser efectivamente iniciada su descarga. De hecho podríamos detectar reintentos, que en este caso no están presentes, si tendríamos varios grupos de puntos separados por una brecha de tiempo para un mismo archivo en una red saturada.

Fig 3.6: Último archivo retornado 3 clientes, 10 archivos y 40 pares de a 10

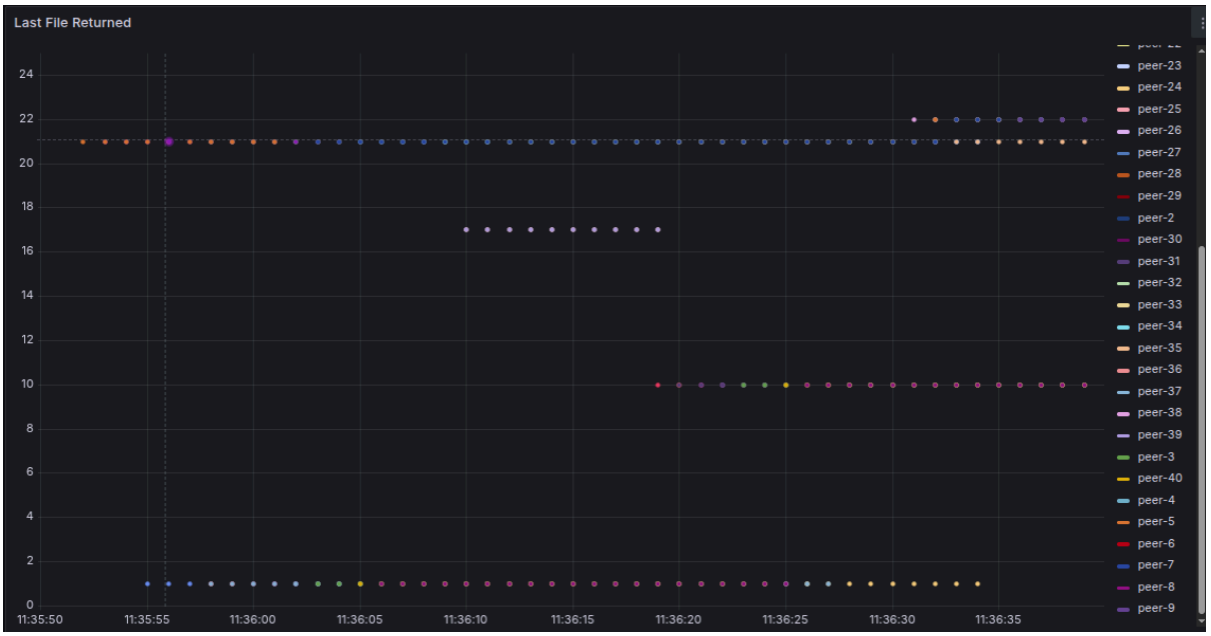


Fig 3.7: Último archivo retornado 3 clientes, 10 archivos y 40 pares de a 10

Los archivos 1 y 21 son los que extienden su actividad durante una mayor franja de tiempo, como se puede observar con mayor detalle en la “figura 3.7”. Esto claramente no se debe a la ocurrencia de reintentos ya que se presenta un sólo tren de puntos de actividad para cada archivo y tampoco se debe a una baja disponibilidad de los archivos en la red, ya que se ha revisado la cantidad de bloques y la replicación de ambos archivos siendo similares al resto. En una inspección más detallada de las métricas provistas por Grafana se observa (en las figuras 3.8 y 3.9) que ambos archivos compiten por obtener bloques de los pares 12

y 24, de manera tal que el culpable no es más que una efímera condición de carrera.

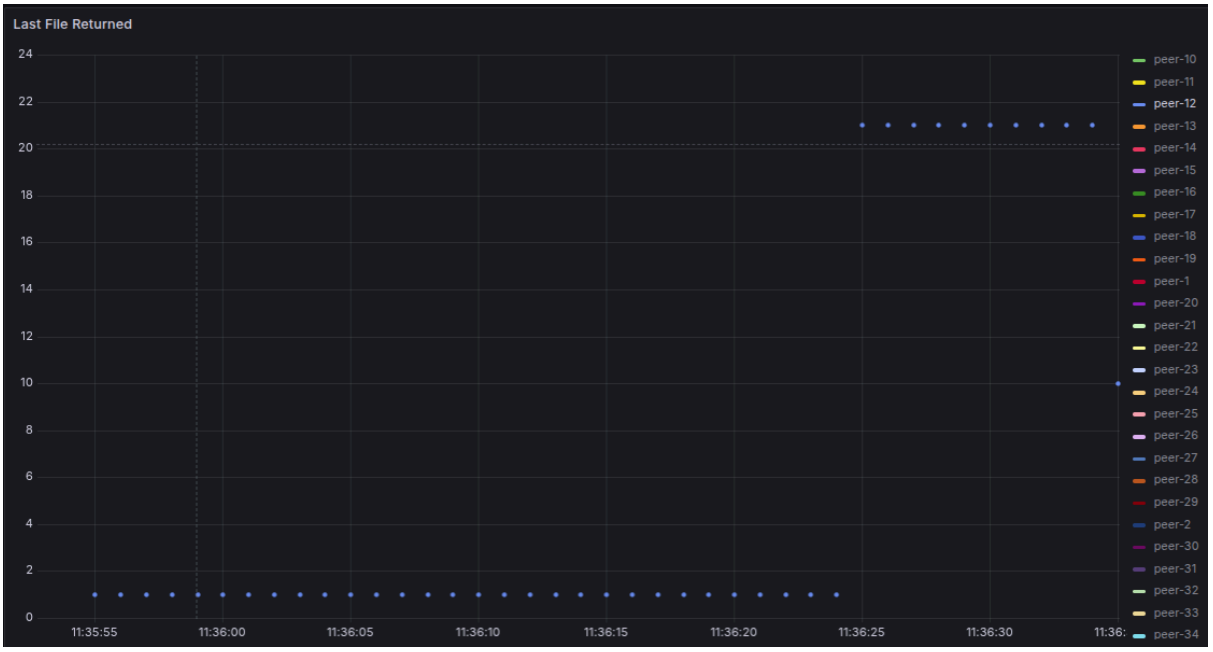


Fig 3.8 (peer-12): Último archivo retornado 3 clientes, 10 archivos y 40 pares agrupados de a 10

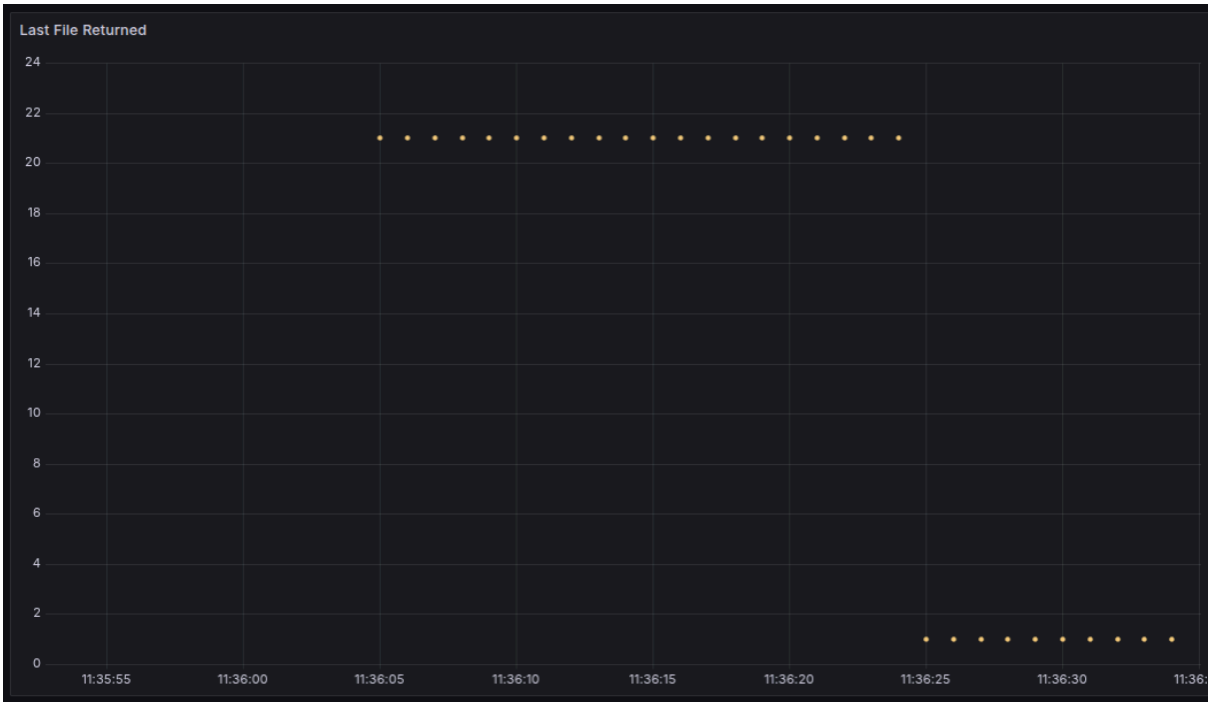


Fig 3.9 (peer-24): Último archivo retornado 3 clientes, 10 archivos y 40 pares agrupados de a 10

Por otro lado, se debe recordar que como el envío de un archivo recuperado desde el par hasta al cliente es una tarea programada, se obtiene un overhead adicional debido a ello.

Rendimiento:

Como se puede observar en la “figura 3.10”, durante la fase de construcción de las bucket-table’s el nodo que más usa el CPU es el peer-1 (57%), el cuál es el bootstrap-node. Esto tiene sentido ya que durante dicho período se debe aumentar la frecuencia de la comunicación con los tres “bootstrap-nodes” secundarios (peer-10, peer-20 y peer-30) los cuales son los siguientes en el ranking de consumo (aproximadamente 48%). Luego el resto de los nodos realizan un uso del CPU por debajo del 20%.

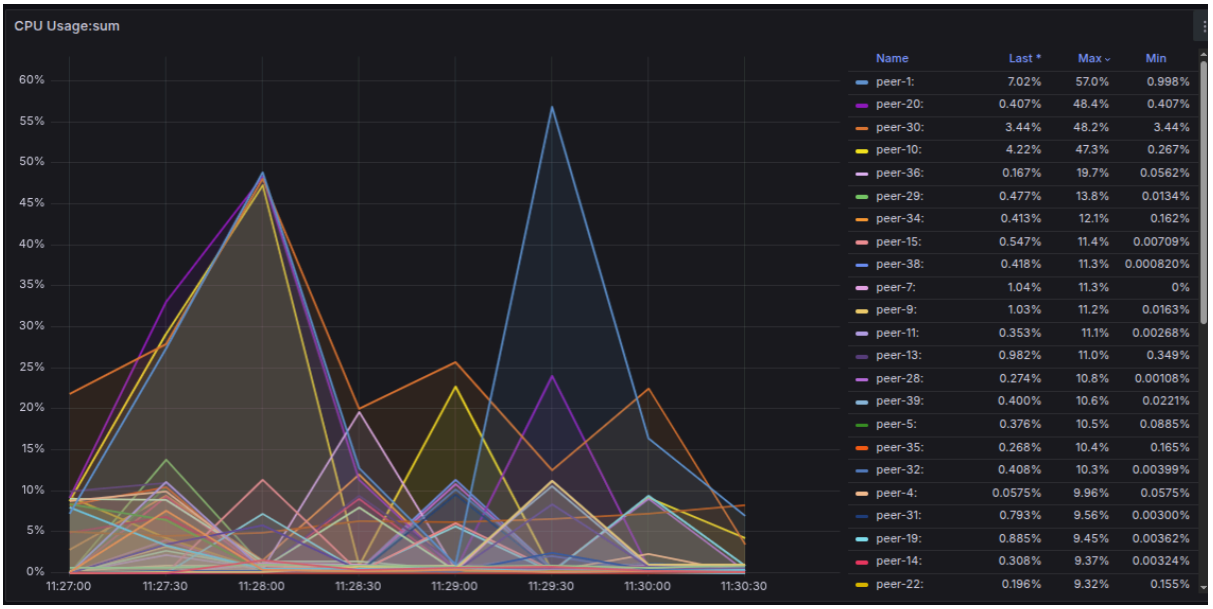


Fig 3.10: Uso de CPU durante fase de construcción bucket-table’s

Si en la “figura 3.11” descartamos el consumo de memoria realizado por Prometheus, Grafana y cAdvisor podemos observar que cada nodo del IFPS no supera un máximo de 60 Mbytes en memoria, llegando a estabilizarse por debajo de los 10 Mbytes. Recordemos que a esta altura aún no se han subido bloques a la red.

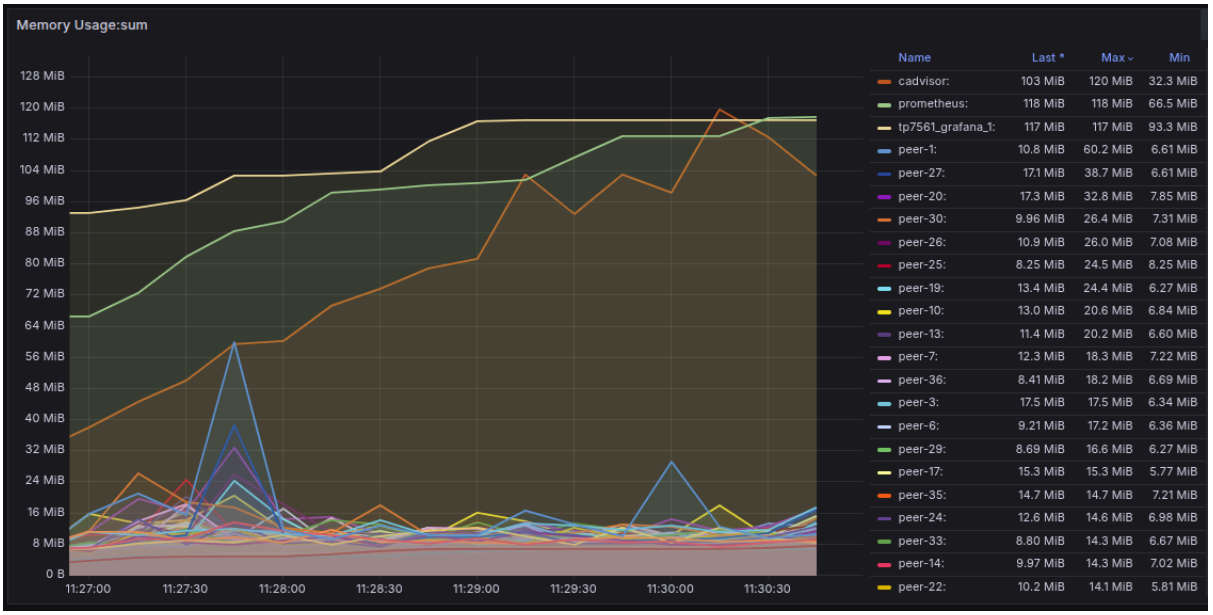


Fig 3.11: Consumo de memoria durante la fase de construcción bucket-table's

Durante la fase de subida y descarga de archivos el consumo de CPU se mantiene por debajo del 2% para la mayoría de los nodos, salvo por breves períodos donde el peer-1 incrementa considerablemente su consumo del CPU. Esto se debe a que el "bootstrap-node" y los "bootstrap-node's" secundarios comparten regularmente los contactos entre sí.

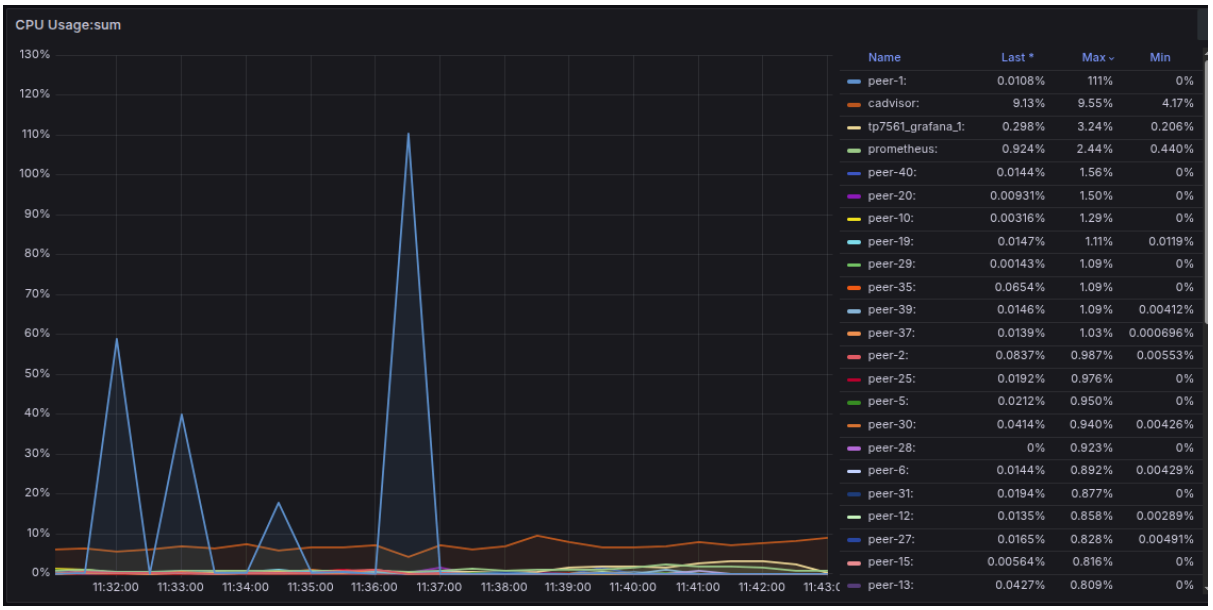


Fig 3.12: Uso de CPU fase de subida y descarga de archivos.

Como se puede observar en la “figura 3.13” el uso de memoria durante esta fase, por parte de los pares, sigue siendo moderado con un máximo de 44 Mbytes.

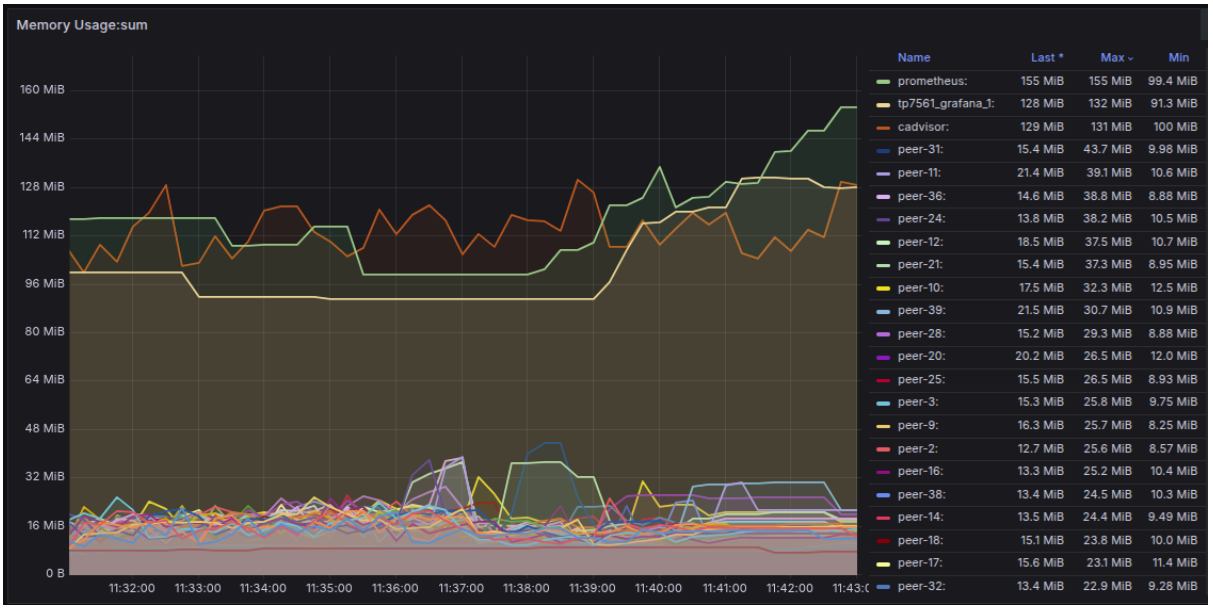


Fig 3.13: Consumo de memoria fase de subida y descarga de archivos.

Como se puede observar en la “figura 3.14” la transmisión de datos en la red IPFS se incrementa drásticamente durante esta fase de subida y descarga de archivos (con un máximo de 250kB/s) como era de esperarse, para caer al mínimo asociado al monitoreo de métricas al finalizar la descarga del último archivo.

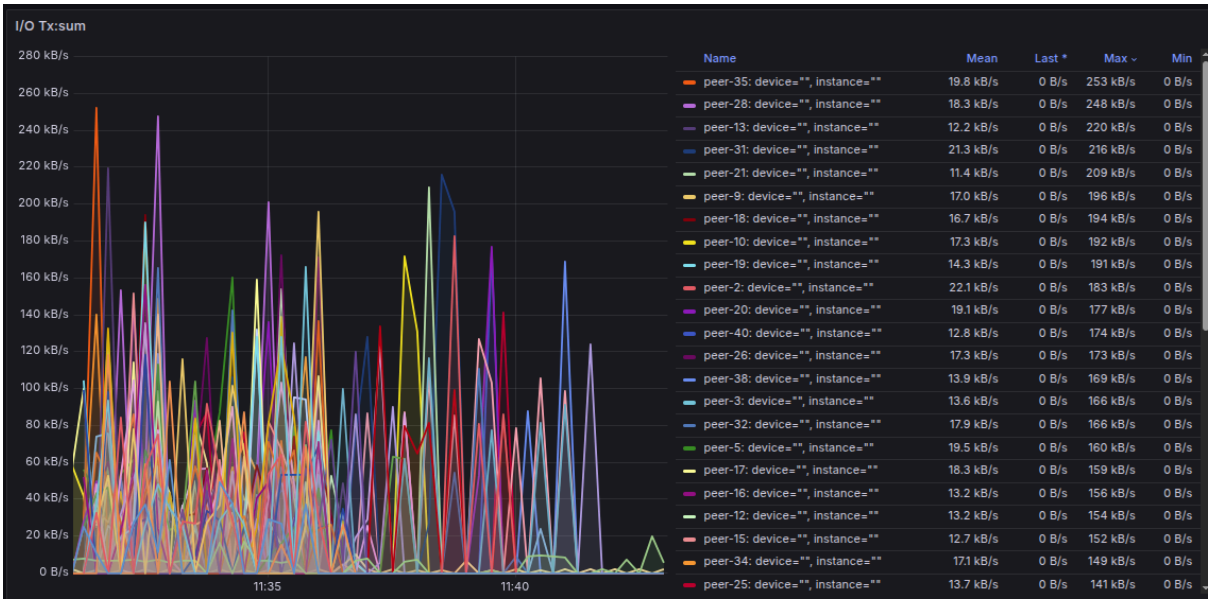


Fig 3.14: Transmisión de datos durante la fase de subida y descarga de archivos.

Estas pruebas se realizaron en una computadora con un procesador Intel Core i3-8100 con 4 núcleos a 3.60GHz bajo el sistema operativo Linux Mint 22.1 Cinnamon.

Conclusión:

Bibliografía:

IPFS: https://ipfs.tech/

¿Qué es y cómo funciona IPFS? Sistema de Archivos Interplanetario: https://www.conquerblocks.com/post/ipfs

Library Genesis: https://libgen.ac/

Library Genesis fue utilizada como fuente de documentos para entrenar a la IA de Meta: https://www.genbeta.com/actualidad/descargar-torrents-portatil-empresa-no-buena-idea-meta-descargo-81-7-tb-libros-copyright-para-su-ia

Audius, una plataforma para subir y moneterizar música: https://docs.audius.org/

Microsoft ION: https://www.criptonoticias.com/comunidad/adopcion/microsoft-lanza-herramienta-de-identidad-basada-en-bitcoin/

Brave lanza una tienda impulsada por Origin: https://medium.com/origin-protocol-spanish/brave-lanza-una-tienda-impulsada-por-origin-7834441ae178

Brave, el primer navegador con el protocolo IPFS integrado: https://www.redeszone.net/noticias/redes/brave-ipfs-navegador-web-distribuida-red-p2p/

Filecoin: https://docs.filecoin.io/basics/what-is-filecoin

Opera para Android compatibilidad con IPFS: https://blog.ipfs.tech/2020-03-30-ipfs-in-opera-for-android/

Partido Pirata Catalán replicó su sitio web en IPFS: https://www.elconfidencial.com/espana/cataluna/2017-09-25/referendum-cataluna-web-clones-burlar-ley_1449241/

Kademlia - a Distributed Hash Table implementation | Paper Dissection and Deep-dive: https://www.youtube.com/watch?v=_kCHOpINA5g

Kademlia, a P2P information system based on XOR metric, MIT: https://pdos.csail.mit.edu/~petar/papers/maymounkov-kademlia-lncs.pdf