Universidad Politécnica
de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Grado en Grado en Ingeniería Informática

Trabajo Fin de Grado

Desarrollo de un Sistema de Intercambio Directo de Archivos entre Dispositivos Basado en IPFS

Autor: Nicolás Cossío Miravalles
Tutor(a): Fernando Pérez Costoya

Madrid, Abril - 2023

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado
Grado en Grado en Ingeniería Informática

Título: Desarrollo de un Sistema de Intercambio Directo de Archivos entre Dispositivos Basado en IPFS

Abril - 2023

Autor: Nicolás Cossío Miravalles
Tutor: Fernando Pérez Costoya
Arquitectura Y Tecnología De Sistemas Informáticos
ETSI Informáticos
Universidad Politécnica de Madrid

Resumen

IPFS, también conocido como Protocolo de Sistema de Archivos Interplanetario, es un protocolo de red y un sistema de archivos diseñado para hacer la web más rápida, segura y abierta. Este sistema permite a los usuarios no solo recibir, sino también alojar contenido en una red P2P completamente descentralizada.

IPFS tiene varias ventajas clave. A diferencia de protocolos como HTTP, en IPFS los recursos se identifican por su contenido en lugar de por su ubicación. Esta característica permite a cualquier nodo de la red convertirse en proveedor de contenido dentro de ella, lo que se traduce en una mayor eficiencia, seguridad, escalabilidad y resiliencia para el almacenamiento y distribución de datos. IPFS facilita la creación de aplicaciones descentralizadas (dApps) al proporcionar herramientas como un sistema de almacenamiento de archivos distribuido y un sistema de nombres descentralizado (IPNS) para la web. Al mismo tiempo, promueve el desarrollo de aplicaciones resistentes a la censura y una web verdaderamente abierta y descentralizada.

Este trabajo de fin de grado se divide en dos partes:

La primera consiste en el estudio del ecosistema de IPFS. Se abarca desde su arquitectura, algoritmo de intercambio de bloques, identificación basada en contenido, hasta su estructura de datos. Se analizan ejemplos de casos de uso en la Web3, como la distribución descentralizada de contenido, el almacenamiento de datos en la cadena de bloques y la publicación de datos permanentes.

La segunda parte del trabajo consiste en la creación de un sistema de intercambio de archivos basado en IPFS. Se presenta un posible diseño de una arquitectura centralizada habitual que se usaría para una aplicación de intercambio seguro de archivos. Se profundiza en posibles puntos únicos de falla, preocupaciones de privacidad y problemas de escalabilidad que surgen al depender de una sola autoridad o servidor. Con estos puntos establecidos, se introduce el sistema ideado. Empleando la naturaleza distribuida de IPFS, esta propuesta tiene como objetivo abordar los problemas mencionados y la mejora de la propiedad y la privacidad de los datos.

Algunas características fundamentales de este sistema son:

- Archivado y compresión de archivos y directorios utilizando tar y gzip.
- Encriptación segura de archivos utilizando aes-256-cbc.
- Encriptación de secretos facilitada por JSON Web Encryption (JWE).
- Verificación de autoría mediante el uso de Identificadores Descentralizados (DIDs), en forma de firma de contenido utilizando JSON Web Signatures (JWS).
- Uso de bases de datos descentralizadas impulsadas por OrbitDB, que permiten:

- Silos de usuarios, registro automático y controladores de acceso distribuidos.
- Notificaciones push mediante una cola de mensajes descentralizada.
- Bases de datos locales con persistencia para uso interno de la aplicación.

La aplicación desarrollada funciona en sistemas operativos Windows, MacOS y Linux. Mediante una interfaz de comandos de consola los usuarios pueden compartir archivos de manera segura y privada sin la necesidad de depender de servidores centralizados.

Abstract

IPFS, also known as the InterPlanetary File System, is a network protocol and file system designed to make the web faster, more secure and open. This system allows users not only to receive but also to host content on a fully decentralized peer-to-peer network.

IPFS has several key advantages. Unlike protocols like HTTP, in IPFS, files are identified by their content rather than their location. This feature allows any node in the network to become a content provider, resulting in greater efficiency, security, scalability and resilience for data storage and distribution.

IPFS facilitates the creation of decentralized applications (dApps) by providing tools such as a distributed file storage system and a decentralized naming system (IPNS) for the web. At the same time it promotes the development of censorship-resistant applications as well as a truly open and decentralized web.

This undergraduate thesis is divided into two parts:

The first part consists of the study of the IPFS ecosystem. From its architecture, block exchange algorithm, content-based addressing, to its data structure. Examples of use cases in Web3, such as decentralized content distribution, blockchain-based data storage, and permanent data publishing, are also analyzed.

The second part of the thesis involves the creation of a secure and decentralized file-sharing system based on IPFS. The process starts by outlining the design and limitations of a typical centralized architecture for an application of the proposed type. Emphasizing on potential single points of failure, privacy concerns and scalability issues that arise from relying on a single authority or server.
With these points established, the devised system is then introduced. Employing the distributed nature of IPFS, this proposal aims to address the aforementioned issues, while also enhancing data privacy and ownership.

Fundamental features of this system encompass:

- File or directory archiving and compression using tar and gzip.
- Secure file encryption using aes-256-cbc.
- Secrets encryption facilitated by JSON Web Encryption (JWE).
- Authorship verification through the usage of Decentralized Identifiers (DIDs), in the form of content signing using JSON Web Signatures (JWS).
- Usage of decentralized databases powered by OrbitDB which enable:
  - User silos, automatic registration, and distributed access controllers.
  - Push notifications via a decentralized message queue.
  - Local databases with persistence for internal application use.

The developed application supports Windows, MacOS, and Linux operating systems. Through a command-line interface, users can securely and privately share files without relying on centralized servers.

Tabla de contenidos

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

El presente Trabajo de Fin de Grado (TFG) se centra en el desarrollo de un sistema de intercambio de ficheros basado en IPFS (InterPlanetary File System)[1].
A continuación, se describen las motivaciones y necesidades que han llevado a la realización de este proyecto.

1.1. Motivación y necesidad

El desarrollo de un sistema de intercambio de ficheros basado en IPFS se encuentra en la confluencia de varias tendencias tecnológicas y sociales que están dando forma al futuro de la web. En particular, este proyecto se relaciona estrechamente con el avance hacia la *Web3*[2], una visión de un internet más descentralizado, seguro y resistente a la censura. En esta sección, exploraremos cómo un sistema de intercambio de archivos encaja en este nuevo panorama y por qué es relevante para el progreso de la Web3.

Los servicios de almacenamiento y compartición de archivos actuales, como Google Drive, Dropbox, Microsoft OneDrive y otros proveedores de almacenamiento en la nube son servicios centralizados. Pese a ser populares y ampliamente utilizados debido a su facilidad de uso, accesibilidad y confiabilidad, presentan ciertos problemas y limitaciones. Los usuarios dependen de una sola entidad para almacenar y gestionar sus archivos, lo que puede generar problemas si la empresa experimenta fallos técnicos, cambia sus políticas de uso, o se convierte en el objetivo de un ataque cibernéticos maliciosos. Además, esto otorga a estas empresas un gran poder sobre los datos de los usuarios, lo que puede conducir a problemas de privacidad y control de la información.

Otras alternativas como FTP (File Transfer Protocol) ofrecen una mayor autonomía y control sobre los archivos, pero también tienen inconvenientes. FTP es un protocolo que permite la transferencia de archivos entre un cliente y un servidor a través de una red. FTP carece de robustas medidas de seguridad modernas, puede ser vulnerable a ataques y requiere un mayor conocimiento técnico y esfuerzo para su configuración y mantenimiento.
En resumen, a pesar de la mayor autonomía y control directo que FTP puede ofrecer, no es comparable con un servicio en la nube en términos de seguridad, facilidad de uso y eficiencia de costos. Esto es teniendo en cuenta los conocimientos y requisitos del usuario promedio de un servicio de estas características.

La arquitectura detrás de este tipo de servicios se basa en el modelo cliente-servidor. En este modelo, un servidor central almacena la información sobre la lista de nodos y recursos disponibles en la red y es vital para el funcionamiento del sistema. Esto facilita encontrar rápidamente los nodos o recursos disponibles, pero el sistema es relativamente vulnerable en términos de fallos o ataques y la escalabilidad está limitada debido a la presión sobre el elemento central [3].

La alternativa a estos servicios centralizados es el uso de tecnologías *peer-to-peer* (de igual a igual en español). Una aplicación peer-to-peer (p2p) es un tipo de red donde no existen clientes ni servidores fijos, sino una serie de nodos que actúan como iguales y pueden funcionar tanto como clientes como servidores entre sí.

Existen varias tecnologías p2p que permiten compartir archivos entre usuarios sin necesidad de un proveedor central, el más famoso y conocido siendo BitTorrent[4]. Sin embargo, estas tecnologías no son adecuadas para el intercambio de archivos entre usuarios no conocidos, ya que requieren que los usuarios confíen en que los archivos que se comparten son los que se anuncian.

Esto es algo que resuelve el Inter Planetary File System (IPFS). El Sistema de Archivos Interplanetario es un sistema de archivos distribuido que busca conectar todos los dispositivos al mismo sistema de archivos. En cierto modo, IPFS es similar a la Web, aunque podría verse como una sola red BitTorrent, intercambiando objetos dentro de un repositorio Git.
En otras palabras, IPFS permite guardar y acceder a bloques de datos identificados por su contenido, no por su ubicación, y que se pueden transferir rápidamente entre los nodos. Además, IPFS usa estos bloques para crear enlaces que también se basan en el contenido, no en una dirección que apunta a una ubicación donde se puede encontrar el contenido. Esto forma un grafo dirigido acíclico generalizado de Merkle (Merkle DAG), una estructura de datos sobre la que se puede construir sistemas de archivos versionados, cadenas de bloques e incluso una Web Permanente. IPFS combina una tabla hash distribuida, intercambio de bloques incentivado y espacio de nombres autocertificante, sin puntos únicos de falla ni necesidad de confianza entre los nodos que la forman[5].

En este proyecto se usará IPFS como bloque central, sobre el que construirá el sistema previamente descrito.

1.2. Objetivos y alcance del proyecto

El objetivo principal de este proyecto es el desarrollo de un sistema de intercambio de ficheros basado en IPFS, mediante una aplicación de escritorio. Este sistema debe permitir a los usuarios compartir archivos de forma segura y confiable, sin necesidad de ningún proveedor central de ningún tipo.
Debe integrar capacidades de encriptación y control de acceso para garantizar la seguridad de los archivos compartidos. La integración de cuentas de usuario, con la posibilidad de hacer grupos, elegir contactos con los que compartir, se propone como algo imprescindible para lograr un sistema autocontenido y sin necesidad de herramientas externas para su uso. Por último se debe integrar un sistema de notificaciones para el que los usuarios puedan recibir avisos de nuevos archivos compartidos, o de cambios en los archivos compartidos.

Para lograr esto se han cumplido los siguientes objetivos:

- Investigar sobre IPFS y su funcionamiento para entender cómo funciona el protocolo y cómo se puede utilizar para el sistema propuesto.
- Investigar sobre el ecosistema en torno a IPFS, con objetivo de comprender la madurez y viabilidad de esta tecnología, así como de las herramientas basadas en esta que se pueden utilizar para el sistema propuesto.
- Diseñar una arquitectura para el sistema de intercambio en torno a las tecnologías y herramientas seleccionadas.
- Implementación de un prototipo funcional del sistema propuesto.
- Analizar la viabilidad de IPFS en base a la experiencia obtenida en el desarrollo del prototipo.
- Analizar posibles mejoras y ampliaciones del sistema propuesto.

Por tanto pese a que el objetivo principal es el desarrollo de un sistema de intercambio de ficheros basado en IPFS, también se realizará una labor de divulgativa sobre IPFS y su ecosistema, con el objetivo de comprender esta tecnología y su viabilidad como alternativa a muchas de las tecnologías actuales.

1.3. Estructura de la memoria

En este capítulo se ha introducido el proyecto, explicando las motivaciones y necesidades que han llevado a su realización.

En el capítulo 2: 'Contexto' se pone en situación el estado actual de tecnologías relacionadas con el proyecto, tanto alternativas como otras implementaciones que usen IPFS u otras tecnologías similares que cumplan parcial o completamente con los objetivos del proyecto. Al comienzo de este capítulo también se explica brevemente la historia de internet y su evolución hasta el presente. La razón de ser de esta sección se debe a la necesidad de poner en situación el porqué detrás de la dominancia de ciertos protocolos que han guiado el modelo de internet actual, y que han llevado a la necesidad de alternativas como IPFS.

Dentro de este capítulo se explica el funcionamiento de IPFS, tratando los siguientes temas: arquitectura interna, funcionamiento, ecosistema y herramientas relacionadas. Con esta sección se busca dar una visión general de esta tecnología y su ecosistema para poder entender el sistema propuesto.

En el capítulo 3: 'Estado del arte' se lleva a cabo un breve análisis de algunas otras implementaciones que usan IPFS o tecnologías similares, así como de otras alternativas a IPFS que cumplen parcial o completamente con los objetivos del proyecto.

El capítulo 4: 'Desarrollo de IPFShare' se centra en el desarrollo del sistema propuesto. Este se ha estructurado en en:

- **Requisitos del sistema:** se explica el funcionamiento deseado del sistema.
- **Diseño del sistema:** se presenta la arquitectura y diseño propuestos en este proyecto, así como las herramientas utilizadas.
- **Implementación:** se explica la implementación realizada, así como las decisiones tomadas durante el desarrollo. Esta sección incluye partes de código relevantes para entender la implementación realizada.

En el capítulo 5: 'Evaluación de la implementación' se realiza una serie de pruebas del sistema desarrollado. Para ello se ha creado un escenario de uso real con distintos usuarios en varios lugares del mundo.

El capítulo 6: 'Resultados y conclusiones' se analiza el resultado obtenido del desarrollo del proyecto. Se contrastará el resultado con los objetivos propuestos y con servicios de transferencia de archivos centralizados.

Sobre el alcance del proyecto, en el capitulo 7: 'Trabajos futuros' se explora las posibles vías de expansión y mejoras para el proyecto en el futuro. También se expresan las esperanzas y expectativas para el crecimiento y posible impacto del mismo.

Capítulo 2

Contexto

En esta sección se intentarán poner en perspectiva, de una forma no exhaustiva, las distintas razones históricas que dan lugar a la necesidad de crear un sistema de almacenamiento descentralizado y distribuido como IPFS. Para ello, se hará un breve repaso histórico de la evolución de Internet y de los protocolos que lo han ido conformando. Posteriormente, se explicará la tendencia centralista del sistema actual, existente a un nivel intrínseco y estructural, además de otros problemas que se derivan de esta situación. Finalmente, se expondrá la propuesta de solución que IPFS ofrece para solventar estos problemas, sobre la que se profundizará en la sección 2.2: 'IPFS como alternativa a HTTP'.

2.1. Breve historia de Internet

2.1.1. Predominancia de los protocolos TCP/IP

La historia de internet está marcada por la competencia entre distintos protocolos de comunicación que buscaban establecerse como el estándar para intercambiar información entre diferentes redes y sistemas. Uno de los episodios más relevantes de esta competencia fue la llamada *"Guerra de los protocolos"* [6], en la que el conjunto de protocolos TCP/IP, creado entre los años 1973 y 1974 por Vint Cerf y Robert Kah, se enfrentó a otras propuestas como OSI, X.25 o SNA[1].

TCP/IP logró imponerse a la competencia debido a las siguientes características principalmente:

- **Interoperabilidad** : La capacidad de TCP/IP para conectarse fácilmente con diferentes tipos de ordenadores y sistemas operativos le otorgaba una ventaja sobre otros protocolos que eran más específicos o limitados en su compatibilidad. Esta característica permitía que diversas tecnologías y plataformas pudieran comunicarse entre sí sin problemas, lo cual era esencial para crear una red global como internet.
- **Flexibilidad** : TCP/IP podía adaptarse a distintos medios de transmisión, como cables de cobre, fibra óptica o incluso enlaces inalámbricos, lo que facilitaba su implementación en una amplia variedad de entornos y situaciones. Otros

[1] En la figura 1 de la página 66 se muestra un resumen de las principales características de cada uno de estos protocolos.

protocolos, en cambio, podrían haber requerido modificaciones o adaptaciones específicas para funcionar en diferentes tipos de medios de transmisión.

- **Resistencia** frente a fallos: TCP/IP fue diseñado para ser robusto en caso de fallos en la red, permitiendo que los paquetes de datos pudieran ser retransmitidos y encontrar rutas alternativas en caso de problemas. Esta capacidad de recuperación era fundamental para garantizar la continuidad y fiabilidad de las comunicaciones en una red global con múltiples nodos y enlaces.
- **Escalabilidad** : TCP/IP podía soportar el crecimiento de la red al permitir la incorporación de nuevos nodos y enlaces sin afectar negativamente su rendimiento. Su diseño jerárquico y descentralizado facilitaba la expansión de la red y evitaba los cuellos de botella que podrían haberse producido con otros protocolos menos escalables.

Estas ventajas hicieron que TCP/IP se convirtiera en la opción preferida frente a otros protocolos, al ser una solución más versátil, resistente y escalable para la creciente demanda de interconexión entre sistemas y redes en todo el mundo. Cabe destacar también que era una solución con arquitectura abierta, no propietaria y de uso gratuito, es decir, sin necesidad de pagar licencias por su uso [7].

Como en toda guerra también hubo un trasfondo político. Este hecho suele ser ignorado al abordarse este tema desde un punto de vista puramente tecnológico. Y es que en 1980, el Departamento de Defensa de Estados Unidos declaró TCP/IP como el estándar para todas las redes militares [8]. A esto se sumaron numerosas comunidades de investigación y universidades que adoptaron TCP/IP como su protocolo de comunicación, como por ejemplo, Stanford University, donde Vint Cerf colaboró con Robert Kahn en el diseño del protocolo [8]; University of California, Los Angeles (UCLA), que participó en el desarrollo temprano y las pruebas de TCP/IP [8]; y University College London (UCL), donde el profesor Peter Kirstein promovió el uso de TCP/IP en Europa y su equipo contribuyó al desarrollo y pruebas del protocolo [9].

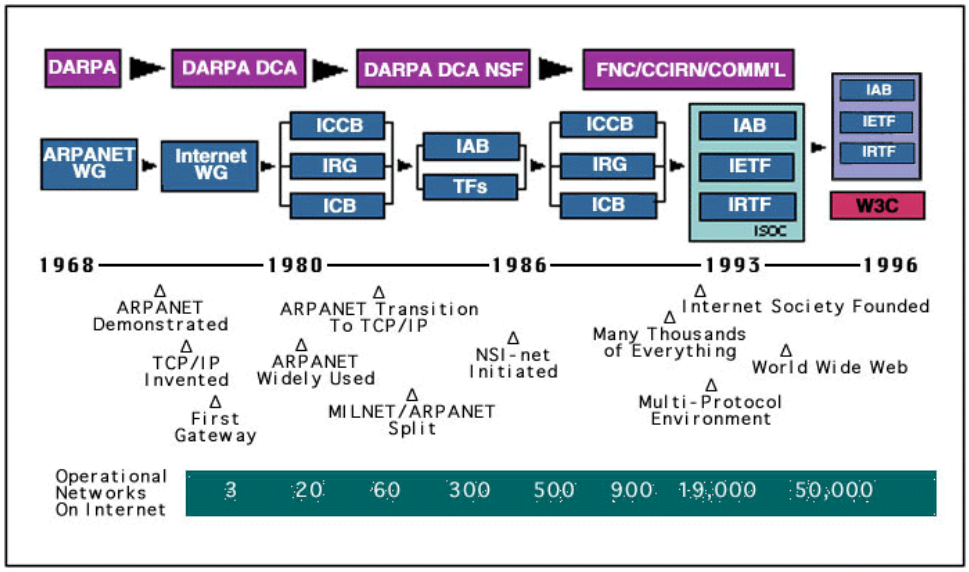

Figura 2.1: Evolución de los protocolos de Internet. Fuente [8]

Esta completa adopción del protocolo se dió por finalizada cuando ARPANET precur-

sor de internet y financiado por la Agencia de Proyectos de Investigación Avanzados de Defensa (DARPA), llevó a cabo la transición exitosa de su antiguo protocolo, el Network Control Program (NCP), a TCP/IP el 1 de enero de 1983 [8].

En resumen, la rápida adopción de la comunidad científica y académica, sumada al respaldo gubernamental consolidaron TCP/IP como el estándar dominante en la industria de las redes de comunicación.

2.1.2. La World Wide Web y HTTP

El modelo TCP/IP asentó una forma de comunicación estándar entre computadores y redes, aunque este estaba limitado principalmente al mundo académico y científico. No fue hasta la creación de la World Wide Web (WWW) cuando el Internet concebido como es en la actualidad se convirtió en un fenómeno global y accesible para todo el mundo.

Antes de la WWW, el acceso a la información en Internet se realizaba a través de los protocolos a nivel de aplicación mostrados en la figura 2.1

Protocolo	**Descripción**
FTP (Protocolo de Transferencia de Archivos)	Utilizado para transferir archivos entre cliente y servidor a través de una red.
Telnet	Basado en texto utilizado para el acceso remoto a computadoras y servidores, permitiendo a los usuarios controlarlos a través de una interfaz de línea de comandos.
Gopher	Diseñado para buscar y recuperar documentos de manera jerárquica, utilizando una interfaz basada en menús.
SMTP (Protocolo Simple de Transferencia de Correo)	Utilizado para enviar mensajes de correo electrónico entre servidores y, finalmente, al cliente de correo del destinatario.
NNTP (Protocolo de Transferencia de Noticias en Red)	Utilizado para la distribución, consulta y recuperación de artículos de noticias en la red Usenet.
POP3 (Protocolo de Oficina de Correos 3)	Utilizado para recuperar mensajes de correo electrónico desde un servidor de correo remoto hasta un cliente de correo local.
IMAP (Protocolo de Acceso a Mensajes de Internet)	Permite a los usuarios acceder y administrar sus mensajes de correo electrónico en un servidor de correo, sin descargarlos a un cliente de correo local.

Cuadro 2.1: Protocolos de capa de aplicación antes de HTTP

Estos servicios se encuentran en nivel de aplicación dentro del *stack* TCP/IP, como se muestra en la figura 2.2. Algunos de ellos se siguen usando hoy en día, o tienen su caso de uso (IMAP, POP3, FTP), pero en lo referente a archivos, ofrecían métodos básicos de navegación y compartición. Carecían de la capacidad de inter-conectar documentos de manera intuitiva y visual. [2]

[2]Cabe destacar que en esta época, los documentos eran principalmente texto plano, sin formato, y

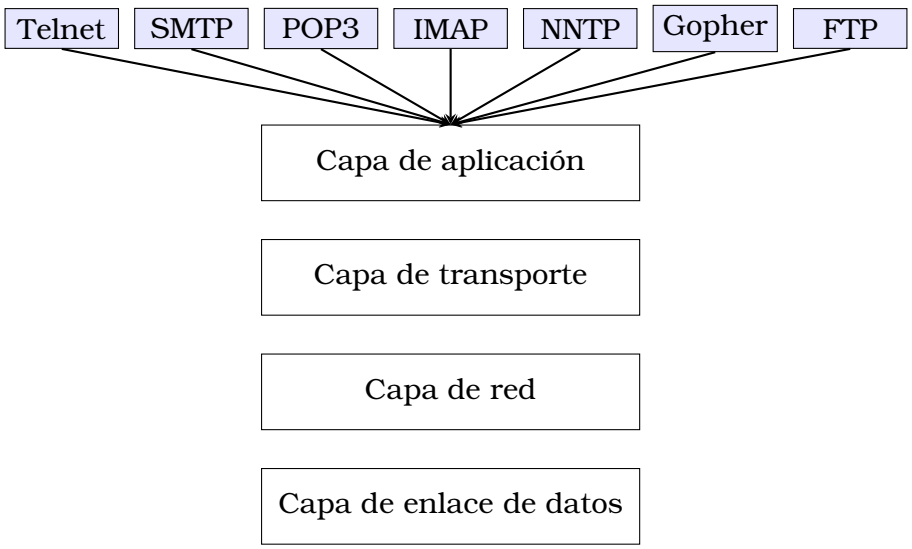

Figura 2.2: Capas del protocolo TCP/IP mostrando algunos protocolos de la capa de aplicación

En 1989, el científico británico Tim Berners-Lee propuso la creación de la WWW, un sistema de información global que permitiría a los usuarios navegar y acceder a documentos interconectados mediante enlaces. Estos documentos, conocidos como páginas web, se almacenarían en computadoras conectadas a la red y podrían ser accedidos a través de un programa especial llamado navegador web, que interpretaría el código de las páginas y mostraría su contenido al usuario.
HTML (Hyper Text Markup Language) es el lenguaje que describe estos documentos. Permite enlazar documentos entre sí mediante hipervínculos. Un hipervínculo es una referencia unidireccional en un documento electrónico que entrelaza diferentes documentos o secciones entre sí. Los usuarios tienen la oportunidad de seguir estos enlaces con tan solo un clic en el texto ancla (texto enlazado) para navegar a los documentos o las secciones correspondientes[10]. Aunque es un concepto simple y con el que cualquier persona en la actualidad esta familiarizada este factor dictamina la forma en la que se usa internet en la actualidad. Los usuarios de internet interactúan con el contenido en internet mediante estos enlaces.
La WWW se basó en tres tecnologías clave: HTML, un lenguaje de marcado para crear páginas web; HTTP, un protocolo para solicitar y transferir recursos a través de la web; y URL, un sistema de direcciones para localizar recursos en la web[8]. Y es este último el que genera una gran problemática que resuelve IPFS.

URL significa Uniform Resource Locator, que se traduce al español como Localizador Uniforme de Recursos. Es un sistema de direcciones utilizado en la web para localizar de manera única recursos como páginas web, imágenes, videos y otros archivos. Una URL consta de varios componentes, incluyendo el esquema (como 'http://' o 'https://'), el nombre de dominio (como 'www.ejemplo.com'), la ruta del recurso y otros parámetros opcionales.

Sin embargo, a medida que la web ha crecido en tamaño y complejidad, el enfoque de direccionamiento basado en la ubicación física de los servidores puede presen-

no existía la posibilidad de incluir imágenes o vídeos.

tar limitaciones. Por ejemplo, si un recurs' se encuentra en una URL específica y esa URL cambia o el servidor deja de estar disponible, el acceso al recurso se verá comprometido.

IPFS aborda este problema mediante el uso de un sistema de direccionamiento basado en el contenido, en lugar de la ubicación. En IPFS, cada archivo y bloque de datos se identifican mediante su contenido, utilizando una función hash criptográfica. Esto permite que los archivos y bloques se puedan encontrar y acceder de forma fiable, independientemente de su ubicación física.

Esto permite a IPFS ofrecer una serie de ventajas sobre el sistema de direccionamiento basado en la ubicación de la web tradicional, como la resistencia a la censura, la persistencia de los datos y la verificabilidad del contenido. En la siguiente sección se profundizará en estas ventajas y en cómo IPFS las hace posibles.

2.2. IPFS como alternativa a HTTP

2.2.1. Introducción

IPFS fue presentado al mundo en 2014 por Juan Benet, en un informe técnico titulado *IPFS - Content Addressed, Versioned, P2P File System*[5]. Benet presenta el concepto de IPFS y su proposición de crear un sistema de archivos distribuido y descentralizado que permita a los usuarios almacenar y compartir archivos de forma segura y confiable.

Benet es también el fundador de Protocol Labs[11], una empresa dedicada a la creación de protocolos de código abierto para la Web3. IPFS es un proyecto de código abierto y, pese a que Protocol Labs está detrás de este, no es el único contribuidor a su desarrollo. Esto es otro de los puntos fuertes de IPFS, la comunidad que lo rodea. En la sección 2.3: 'Ecosistema en torno a IPFS' se profundiza en este aspecto.

En IPFS, cada archivo se identifica de manera única a través de su contenido mediante un hash criptográfico. Esto significa que cualquier nodo en la red puede actuar como un proveedor de contenido al almacenar y compartir archivos, permitiendo una mayor disponibilidad y un internet verdaderamente descentralizado. En lugar de depender de un único servidor web para acceder a un recurso, los usuarios pueden obtener el contenido de cualquier nodo que tenga ese recurso en particular.

Estos identificadores de contenido se conocen como CID (Content Identifier). Dado que un CID es un puntero que señala a un contenido particular, se puede usar un CID en vez de URL en un enlace. De esta manera se puede acceder a un recurso de manera fiable, independientemente de su ubicación física, mientras haya algún otro nodo de la red en posesión del contenido que buscamos.

2.2.2. Fundamentos

IPFS opera a través de tres principios fundamentales que marcan una diferencia significativa con respecto a los sistemas de archivos convencionales: direccionamiento por contenido, red peer-to-peer y el grafo acíclico dirigido de Merkle (Merkle DAG).

Direccionamiento por Contenido: En IPFS, los archivos no se ubican por su dirección sino por su contenido. Cada archivo posee un identificador único, denominado CID (Content Identifier), generado a partir de un hash criptográfico de su contenido. Esta característica asegura la inmutabilidad de los archivos, es decir, los archivos no pueden ser alterados sin modificar su CID. Adicionalmente, el direccionamiento por contenido favorece la deduplicación, dado que archivos con contenido idéntico compartirán el mismo CID, lo que conlleva a su almacenamiento único dentro de la red.

```
$ ipfs add somefile # comando para añadir archivos a ipfs a través de la
↪  línea de comandos
added QmVtuHo6C7NUonYTYUNbmgGxvSwrTVGXuahJxhxnoSxPpM somefile
```

Figura 2.3: Ejemplo de CID generado por IPFS

Red Peer-to-Peer: IPFS se basa en una red descentralizada en la que cada integrante, o nodo, puede interactuar directamente con cualquier otro nodo, sin la necesidad de intermediarios o servidores centrales. Los nodos funcionan tanto como proveedores como consumidores de contenido, guardando y compartiendo fragmentos de archivos con otros nodos. Esta red peer-to-peer hace que el contenido sea más accesible y resistente a la censura, al evitar la existencia de un único punto de fallo o control.

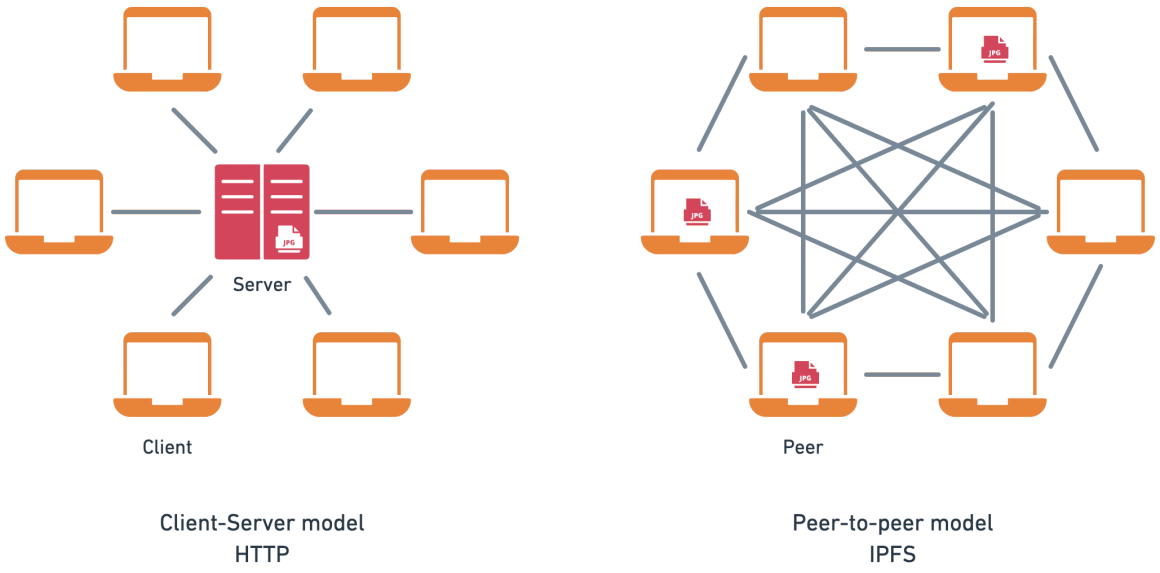

Figura 2.4: Red centralizada en comparación con una red descentralizada

Grafo Acíclico Dirigido de Merkle (Merkle DAG): Los archivos y sus relaciones dentro de IPFS se representan mediante una estructura de datos conocida como Merkle DAG. Un Merkle DAG es un grafo donde cada nodo tiene un identificador único (CID) que se genera a partir de su contenido y el de sus nodos hijos. Los nodos pueden ser hojas o nodos intermedios, dependiendo de si tienen o no nodos hijos. Los nodos hoja contienen datos binarios de los archivos, mientras que los nodos intermedios contienen enlaces a otros nodos. Los nodos intermedios permiten dividir archivos grandes en bloques más pequeños y formar estructuras jerárquicas, como

directorios o sistemas de archivos. El Merkle DAG facilita la verificación de integridad y autenticidad de los archivos, dado que cualquier cambio en el contenido o en los enlaces se refleja en el CID del nodo afectado y sus ancestros.

Cada uno de estos conceptos se profundizará dentro del apartado correspondiente a continuación.

2.2.3. Arquitectura

IPFS es un conjunto de protocolos de código abierto que combina múltiples conceptos existentes de redes peer-to-peer (P2P), datos enlazados y otras áreas para permitir que los participantes intercambien fragmentos de archivos.

Estos conceptos concretados en protocolos forman distintos niveles de abstracción, cada uno de los cuales se puede utilizar de forma independiente y conforman la arquitectura de IPFS, también conocido como el *stack* de protocolos de IPFS. En la figura 2.5 se muestra el stack de protocolos de IPFS.

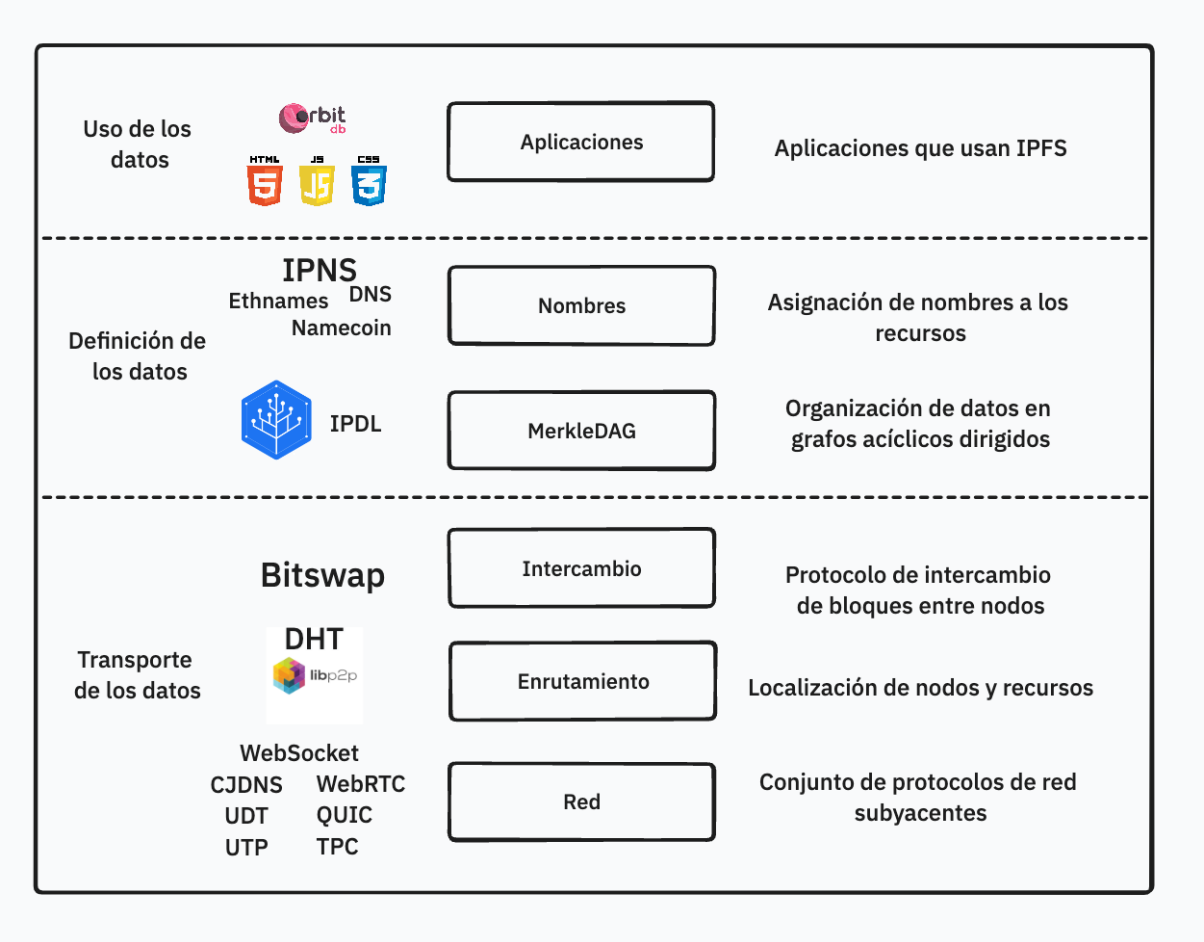

Figura 2.5: Stack de protocolos IPFS

Como se puede observar, este stack se divide tres grupos según la funcionalidad que brinda cada capa. Este diseño en capas subdivido en componentes independientes permite que estos pueden ser ampliados o reemplazados según se necesite. Esta

modularidad en el diseño está respaldada por una biblioteca de redes P2P llamada libp2p[12].

Libp2p es una suite de protocolos y herramientas modulares que permite la creación de sistemas de red peer-to-peer (P2P). Se encarga de gestionar todas las necesidades de red, como la negociación de protocolos, el enrutamiento, la detección de nodos y la transmisión de datos.

2.2.3.1. Capa de red

En la capa de red encontramos los protocolos de transporte de red mediante los cuales los nodos se pueden comunicar. Estos protocolos provienen de libp2p[13] y son los siguientes:

- **TCP**: proporciona una entrega de datos confiable, ordenada y con control de errores sobre redes IP.
- **UDP**: proporciona una entrega de datos simple, sin conexión y no confiable sobre redes IP.
- **QUIC**: Un protocolo de transporte multiplexado y seguro que se ejecuta sobre UDP, proporcionando flujos confiables, de baja latencia y cifrados.
- **WebSockets**: Un protocolo que permite la comunicación bidireccional entre un navegador web y un servidor sobre TCP.
- **WebRTC**: Un protocolo que permite la comunicación en tiempo real entre navegadores web mediante conexiones peer-to-peer.

Debido a esta gran variedad de protocolos de transporte, libp2p proporciona una forma de identificar el transporte que se esté usando mediante direcciones *multiaddr*. Los multiaddresses son una forma de representar las direcciones de red como encapsulaciones de protocolos arbitrarios. Estos multiaddresses admiten direccionar para cualquier protocolo de red. Siguen una sintáxis simple, lo que los hace fáciles de analizar y construir.

En IPFS, se utilizan los multiaddresses para identificar y localizar los nodos en la red. Cada nodo al configurarse por primera vez genera un par de claves pública y privada. La clave pública se utiliza para crear un identificador único del nodo en la red llamado peerID, que es el hash de su esta clave pública. La clave privada se utiliza para firmar mensajes y autenticar la identidad del nodo. Además, los nodos tienen una o más direcciones de red que combinan protocolos y valores para indicar cómo conectarse a ellos. Por ejemplo, una dirección de red podría ser /ip4/1.2.3.4/tcp/4001/ipfs/QmFoo, lo que significa que el nodo QmFoo está escuchando conexiones TCP en el puerto 4001 utilizando la dirección IP 1.2.3.4.

Los multiaddresses se pueden encapsular entre sí para crear capas de transporte más complejas. Por ejemplo, se puede utilizar /dns4/example.com/tcp/1234/tls/ws/tls para indicar una conexión segura con WebSockets sobre TLS utilizando el dominio example.com y el puerto 1234.

La amplia variedad de protocolos de transporte disponibles garantiza la adaptabilidad de IPFS, ya que los nodos pueden utilizar múltiples protocolos de transporte simultáneamente y cambiar entre ellos según las condiciones de la red. Esto significa que

las opciones de transporte de un nodo dependen del entorno en el que se ejecute. Por ejemplo, en un navegador web sólo se pueden utilizar WebSockets y WebRTC, mientras que en otros entornos se pueden utilizar todos los protocolos mencionados anteriormente, siempre que sean compatibles a nivel de sistema operativo y hardware.

2.2.3.2. Enrutamiento y descubrimiento de nodos

Descubrimiento de nodos:
Es el proceso de encontrar y anunciar servicios a otros nodos en una red P2P. Se puede realizar utilizando diversos protocolos, como por ejemplo, la difusión de mensajes a todos los nodos de la red o utilizar una serie nodo de arranque para proporcionar una lista de nodos conocidos.

Esto último se conoce como nodos de arranque (bootstrapping). Es una lista de nodos predefinidos y de confianza que ayudan a los nuevos nodos a unirse a la red y a descubrir otros nodos, facilitando el proceso de construcción y mantenimiento de la red distribuida. La lista de bootsrappers la define cada nodo. La figura 2.6 muestra los bootstrappers por defecto en una instalación de IPFS.

```
"Bootstrap": [
"/dnsaddr/bootstrap.libp2p.io/p2p/QmNnooDu7bfjPFoTZYxMNLWUQJyrVwtbZg5gBMjTezGAJN",
"/dnsaddr/bootstrap.libp2p.io/p2p/QmQCU2EcMqAqQPR2i9bChDtGNJchTbq5TbXJJ16u19uLTa",
"/dnsaddr/bootstrap.libp2p.io/p2p/QmbLHAnMoJPWSCR5Zhtx6BHJX9KiKNN6tpvbUcqanj75Nb",
"/dnsaddr/bootstrap.libp2p.io/p2p/QmcZf59bWwK5XFi76CZX8cbJ4BhTzzA3gU1ZjYZcYW3dwt",
"/ip4/104.131.131.82/tcp/4001/p2p/QmaCpDMGvV2BGHeYERUEnRQAwe3N8SzbUtfsmvsqQLuvuJ",
"/ip4/104.131.131.82/udp/4001/quic/p2p/QmaCpDMGvV2BGHeYERUEnRQAwe3N8SzbUtfsmvsqQLuvuJ"
],
```

Figura 2.6: Bootstrappers por defecto en una instalación de IPFS

Enrutamiento:
Por otro lado, enrutamiento se refiere a encontrar la ubicación específica de otro nodo de la red. Esto se realiza típicamente mediante el mantenimiento de una tabla de enrutamiento u otra estructura de datos similar que realiza un seguimiento de la topología de la red. En el caso de IPFS se usa una tabla de hash distribuida conocida como DHT (Distributed hash table).

En la práctica, la distinción entre el enrutamiento y el descubrimiento de nodos no siempre está clara, de hecho suelen ocurrir simultáneamente.

Los protocolos principales[3] que usa IPFS y libp2p para este propósito son:

- **ping**: Protocolo de comprobación de disponibilidad. Los nodos pueden utilizarlo para verificar la conectividad y el rendimiento entre ellos.
- **autonat**: Protocolo de detección de NAT. Asiste a los nodos en la identificación de su accesibilidad desde internet, esto es útil para detectar si los nodos que se encuentran ocultos detrás de un NAT o de un firewall [14].
- **identify**: Protocolo para el intercambio de claves y direcciones con otros nodos.

[3]Existen más protocolos de enrutamiento y descubrimiento de nodos en libp2p pero estos son los más utilizados por IPFS.

Facilita el intercambio de información esencial, como los protocolos soportados, las claves públicas, las direcciones, etc.
- **kademlia**: Protocolo para la implementación de una tabla hash distribuida para el almacenamiento descentralizado y la recuperación de información de nodos y contenidos.
- **mdns**: Protocolo de descubrimiento de nodos locales con cero configuración, usando DNS de multidifusión. Ofrece un mecanismo para que los nodos en la misma red local se descubran entre sí sin configuración previa.
- **Circuit Relays**: Es un protocolo que facilita a los nodos el reenvío de tráfico en nombre de otros nodos que no tienen un acceso directo entre ellos[15].
- **rendezvous**: Un protocolo de encuentro que se utiliza como un punto común entre dos rutas. Los puntos de encuentro son típicamente nodos que están bien conectados y son estables en una red, y pueden manejar grandes cantidades de tráfico y datos. Sirven como un centro para que los nodos se descubran. De los pocos mecanismos centralizados que usa libp2p [16].
- **pubsub**: Es una interfaz PubSub para libp2p, diseñada para establecer una base para la comunicación de mensajes mediante un patrón de publicación y suscripción entre los nodos de la red libp2p. Existen diferentes implementaciones de este protocolo, como FloodSub, GossipSub que proporcionan diferentes ventajas.

Tal como se indicó previamente, libp2p dispone de varias estrategias para el enrutamiento y la detección de nodos. IPFS las utiliza todas en sus diferentes configuraciones. La elección de la combinación de estas estrategias se realiza en función del contexto particular de un nodo respecto de otros nodos y la red.

2.2.3.3. Mecanismo de intercambio de contenido

Bitswap: Es un protocolo de intercambio de contenido que se ejecuta sobre una red P2P. Bitswap permite a los nodos intercambiar bloques de datos entre los nodos que conforman la red. Estos pueden solicitar bloques de datos a otros nodos y compartir los bloques que disponen. Bitswap utiliza un mecanismo de intercambio de deuda para garantizar que los nodos intercambien bloques de datos de manera justa y equitativa. Todo esto es posible gracias a que Bitswap mantiene un registro de los bloques que cada nodo tiene y los bloques que necesita.

La transferencia de datos (bloques) en IPFS está inspirada en BitTorrent, pero no es igual uno a uno comparado con este. Dos características de BitTorrent que utiliza IPFS:

- Estrategia de tit-for-tat (el que no comparte no recibe).
- Obtén primero las piezas raras (mejora el rendimiento).

Una diferencia notable es que en BitTorrent cada archivo tiene un enjambre (también conocido como *swarm*) separado de nodos (formando una red P2P entre ellos). En cambio, IPFS es una única red de nodos formando un gran swarm. La variedad de BitTorrent en IPFS es el ya mecionado Bitswap.

Bitswap es el algoritmo de intercambio de bloques, pero para realizar este intercambio primero se debe saber qué nodos pueden proveer los bloque que se buscan. Esto es

posible gracias a la DHT. En IPFS la DHT se utiliza principalmente con dos funciones:

1. **Enrutamiento:** se ha explicado en la subsección anterior.
2. **Anuncios de provisión/consumición de contenido:** los nodos publican en la DHT los bloques de datos que tienen disponibles. Esto permite a otros nodos saber quién tiene los bloques de datos que están buscando. Esta tabla de hash se distribuye por la red mediante el algoritmo de Kademlia.

Sobre el segundo, que es el concierne dentro del mecanismo de intercambio de contenido: Cada nodo tiene dos lista de bloques (CIDs). Bloques que posee y puede proporcionar, y bloques que desea obtener.

- Al recibir una lista de deseos, una entidad que use Bitswap debería procesarla eventualmente y responder al solicitante con información sobre el bloque o el bloque en sí.
- Al recibir bloques, el nodo consumidor debe enviar una notificación de cancelación al resto de nodos a los que ha pedido estos bloques, señalando que ya no los desea.

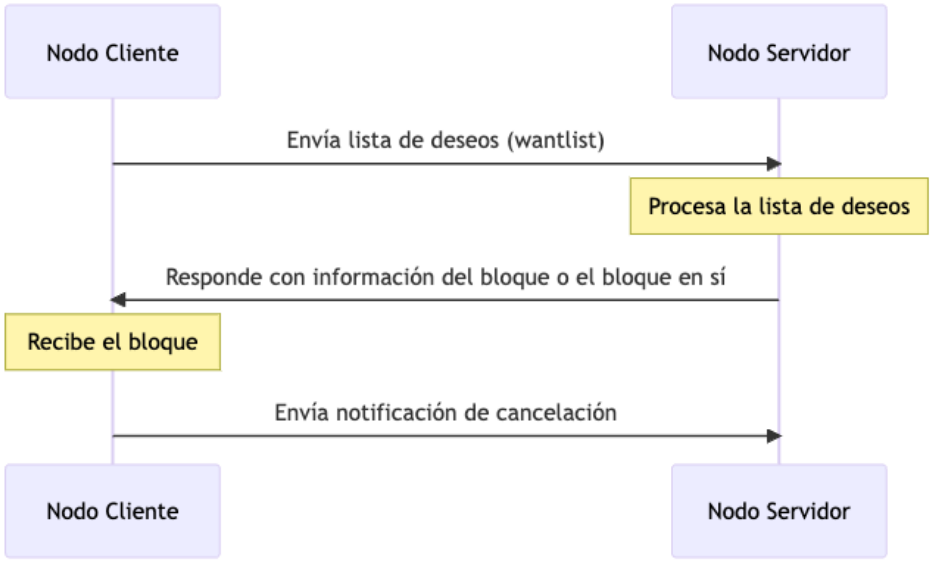

Figura 2.7: Protocolo Bitswap

Por último cabe destacar un concepto muy importante dentro de IPFS en torno al guardado de bloques: **la recolección de basura.**

Cada nodo tiene una capacidad de almacenamiento, siendo esta el límite de bloques que puede almacenar. A medida que un nodo va obteniendo y compartiendo más contenido a través de IPFS, los bloques que recibe se van ubicando su almacenamiento local. La cantidad de bloques puede aumentar rápidamente y ocupar espacio innecesario. Algunos de estos bloques pueden estar referenciados por objetos obsoletos o que ya no son necesarios, lo que significa que no se utilizan ni se acceden directamente.

La recolección de basura en IPFS es el proceso mediante el cual se eliminan los bloques que ya no son necesarios en un nodo. Sin embargo, IPFS utiliza un sistema de almacenamiento basado en referencias, lo que significa que un bloque puede ser referenciado por múltiples objetos y mantenerse en el sistema aunque no esté directamente en uso.

Para evitar que los bloques necesarios sean eliminados por accidente sin el deseo del usuario poseedor del nodo, IPFS introduce el concepto de *pinning*. Un pin es una instrucción que le indica al nodo que debe mantener un bloque o conjunto de bloques en su almacenamiento local, incluso si no están siendo utilizados directamente por el nodo.

Los pinsets son conjuntos de CIDs que se desean mantener en el nodo. Estos pinsets permiten a los usuarios especificar qué bloques desean mantener de manera persistente, evitando así que sean eliminados durante el proceso de recolección de basura.

En resumen, los pinsets son conjuntos de CIDs que representan bloques que un nodo desea mantener en su almacenamiento local, y utilizan el concepto de 'pins' para asegurarse de que estos bloques no sean eliminados accidentalmente durante la recolección de basura.

2.2.4. Modelo de datos

2.2.4.1. DAG de Merkle

El modelo de datos en IPFS está basado en árboles de Merkle. Es una estructura de datos en la que cada nodo es una representación hash de un conjunto de datos. Los nodos hoja son representaciones (generalmente a través de una función de hash) de bloques de datos, mientras que cada nodo interno es la representación (nuevamente, generalmente a través de una función de hash) de sus nodos hijos. Esto crea un sistema en el que cualquier cambio en los datos originales cambiará los hashes en la ruta hasta la raíz del árbol, proporcionando una forma de verificar la integridad de los datos.

Un DAG de Merkle es una estructura donde cada nodo es un árbol de Merkle, y están conectados formando un grafo acíclico direccionado. Esto significa que los nodos están conectados de tal manera que siempre hay una dirección (de nodos padres a nodos hijos) y no hay ciclos (no puedes empezar en un nodo, seguir las conexiones y volver al nodo original).

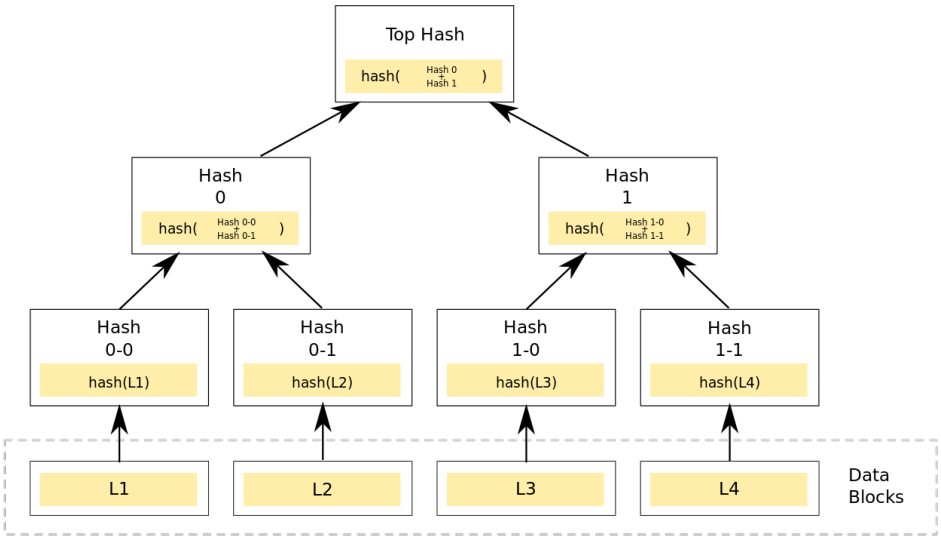

Figura 2.8: Ejemplo de un árbol de Merkle

2.2.4.2. IPLD

IPLD (InterPlanetary Linked Data)[17] es un estructura basada en un DAG de Merkle que permite enlazar datos entre diferentes sistemas distribuidos, como IPFS, Bitcoin o Ethereum. Una estructura de datos inmutable es aquella que no puede ser modificada una vez creada, lo que ofrece ventajas como la seguridad, la consistencia y la ausencia de efectos secundarios. IPLD utiliza estructuras de datos inmutables para representar los bloques de datos que se almacenan y se enlazan entre sí mediante CIDs.

IPLD es una capa de abstracción que permite a los desarrolladores trabajar con datos en diferentes plataformas y protocolos como si estuvieran trabajando con un solo sistema cohesivo. Permite la interoperabilidad a gran escala entre diferentes sistemas de almacenamiento de datos, lo que facilita la creación de aplicaciones y servicios más robustos y resistentes en entornos distribuidos.

El DAG también garantiza a IPFS con característica de control de versiones como Git. Aunque este es un apartado en el que se profundiza poco en IPFS. En el whitepaper, Benet se refiere más bien al hecho de que al igual que Git, IPFS usa Merkle DAGs, y por lo tanto posee características de control de versiones. Los nodos en un Merkle DAG son inmutables. Cualquier cambio en un nodo alteraría su identificador y, por lo tanto, afectaría a todos los ascendentes en el DAG, creando esencialmente un DAG diferente.

En lo que respecta a este trabajo, este hecho permite la propia existencia de OrbitDB, el cual es una pieza clave en el desarrollo de este proyecto. OrbitDB implementa bases de datos distrubuidas en IPFS, se basa en este concepto al ser los datos inmutables y permitir la reconstrucción de la base de datos mediante el intercambio de objetos y la actualización de referencias remotas. TODO poner referencia a orbitdb

2.2.4.3. Códecs de IPLD

Los códecs de IPLD son funciones que transforman el modelo de datos de IPLD en bytes serializados para que puedas enviar y compartir datos, y transforman los bytes serializados de nuevo en el modelo de datos de IPLD para que puedas trabajar con él. Algunos de estos códecs incluyen:

- **DAG-CBOR:** es un formato binario que soporta el modelo de datos de IPLD al completo. Ofrece un excelente rendimiento y es adecuado para cualquier tipo de trabajo.
- **DAG-JSON:** está basado en JSON (Javascript Object Notation). Es un formato más legible, lo que lo hace muy conveniente para la interoperabilidad, el desarrollo y a la hora de depurar código que haga uso de IPLD.
- **DAG-PB:** es otro formato binario usado principalmente para serializar datos en fornmato de unixfsv1 (ir a 2.2.4.4: 'Unixfs' para más información).
- **DAG-JOSE:** El codec dag-jose es un formato para firmar y cifrar objetos JSON.

En este proyecto se ha hecho especial incapié en el uso de DAG-JOSE como códec de los datos en el DAG de Merkle. Esto permite la firma y cifrado de en objetos JWS (JSON Web Signatures) y JWE (JSON Web Encryption) representados en nodos del DAG de IPLD. TODO poner referencia a la parte de JWE y JWS

2.2.4.4. Unixfs

Sobre todo este modelo de datos se establecen otras estructura o abstracciones como Unixfs.

Cuando se agrega un archivo a IPFS, puede que sea demasiado grande para caber en un solo bloque, por lo que se divide en disintos bloques que luego son representados mediante metadatos en una lista de enlaces a estos bloques. UnixFS es un formato usado para describir archivos, directorios y enlaces simbólicos en IPFS. Este formato de datos se usa para representar archivos y todos sus enlaces y metadatos en IPFS. UnixFS crea un bloque (o un árbol de bloques) de objetos enlazados.

```
$ ipfs add Pictures/Wallpapers/ -r
added QmT4pCxAKwjKz58FRkKGZUkuk9dBogzvSqfKaP9aXxRsLh Wallpapers/flow 1.jpg
added QmY1KMKAoHmDK9V3woMGGarVUJ4a1a4HCW1CxRXMtC9KwP Wallpapers/flow 2.jpg

added QmYmdHatxVXfSy9Gjdp8e75cL2mvsGyZ1tT6Qo3ijUV2h5 Wallpapers
```

Al obervar el contenido del CID final de la carpeta subida en el DAG

```
$ ipfs dag get /ipfs/QmYmdHatxVXfSy9Gjdp8e75cL2mvsGyZ1tT6Qo3ijUV2h5 | jq
```

```
{
  "Data": {
    "/": {
      "bytes": "CAE"
    }
  },
  "Links": [
```

```
      {
      "Hash": {
        "/": "QmT4pCxAKwjKz58FRkKGZUkuk9dBogzvSqfKaP9aXxRsLh"
      },
      "Name": "flow 1.jpg",
      "Tsize": 50087191
    },
    {
      "Hash": {
        "/": "QmVRXnTfUDxioWNZ5FbA79xuiZGtkUuxL6rae1MtstGuzu"
      },
      "Name": "flow 2.jpg",
      "Tsize": 32032025
    },
  ]
}
```

Como se puede observar el DAG contiene la estructura de datos que modela un directorio, en este caso de tipo directorio.

```
$ ipfs files stat /ipfs/QmYmdHatxVXfSy9Gjdp8e75cL2mvsGyZ1tT6Qo3ijUV2h5
Size: 0
CumulativeSize: 549538133
ChildBlocks: 22
Type: directory
```

En cambio si se observa el contenido en el DAG de uno de los archivos enlazados:

```
$ ipfs dag get QmT4pCxAKwjKz58FRkKGZUkuk9dBogzvSqfKaP9aXxRsLh | jq
{
  "Data": {
    "/": {
      "bytes": "CAIYnKzwFyCAgOAVIJyskAI"
    }
  },
  "Links": [
    {
      "Hash": {
        "/": "QmbhvAYKs4ERhnvujqDdTVCZdV2Y8UqrnFjGTVver1rzFL"
      },
      "Name": "",
      "Tsize": 45623854
    },
    {
      "Hash": {
        "/": "QmRw866oefrQ98iYb71cvKMmHTLsoTUW5j2cmRR5kkRqLf"
      },
      "Name": "",
      "Tsize": 4463228
    }
  ]
}
```

El archivo ocupa dos bloques cuyos CIDs están en la lista de enlaces del objeto.

```
$ ipfs files stat /ipfs/QmT4pCxAKwjKz58FRkKGZUkuk9dBogzvSqfKaP9aXxRsLh
QmT4pCxAKwjKz58FRkKGZUkuk9dBogzvSqfKaP9aXxRsLh
Size: 50075164
CumulativeSize: 50087191
```

```
5 ChildBlocks: 2
6 Type: fil
```

2.2.4.5. MFS

Sobre Unixfs existe otra capa más que es la que realmente permite una interacción con IPFS como si de un sistema de ficheros tradicional se tratara. Este componente es denominado *Mutable File System* (MFS) o Sistema de Archivos Mutable. Para hacer esto posible, MFS mantiene un mapa de la estructura de archivos y directorios en IPFS. Cada vez que se realiza una operación en MFS, como crear o mover un archivo, se actualiza este mapa. Sin embargo, los datos subyacentes en IPFS permanecen inalterados. Esto significa que se pueden cambiar la estructura y organización de archivos y directorios en MFS sin tener que copiar o mover los datos reales, manipulando enlaces dentro del DAG.

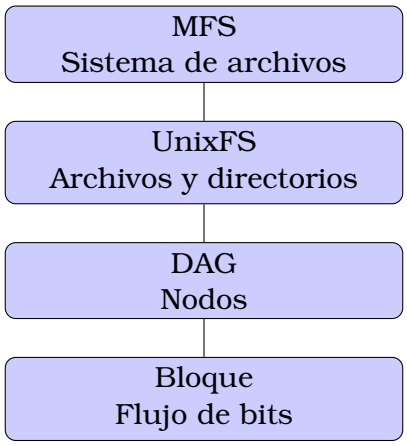

Figura 2.9: Capas de abstracción sobre los datos en IPFS

2.2.5. Sistema de nombres

IPNS, o Sistema de Nombres Interplanetario, es un componente fundamental de IPFS que permite la creación de nombres persistentes para diferentes nodos en la red IPFS. Dado que los contenidos en IPFS son inmutables y se accede a ellos a través de sus CIDs, cualquier cambio en el contenido dará lugar a un nuevo CID. Esto puede resultar inconveniente para los usuarios que necesitan referirse a un contenido específico, incluso si este cambia con el tiempo. Aquí es donde entra en juego IPNS.

IPNS proporciona una capa de indirección que permite a los usuarios referirse a contenidos que pueden cambiar con el tiempo usando un nombre persistente. En lugar de tener que actualizar el CID cada vez que cambia el contenido, los usuarios pueden referirse al contenido usando un identificador IPNS. Este identificador es una clave criptográfica que se genera cuando se inicializa un nodo IPFS, y es única para cada nodo.

Cuando se desea publicar contenido bajo IPNS, se crea un registro que vincula el identificador IPNS con el CID del contenido. Este registro se firma con la clave privada

del nodo, garantizando que solo el propietario del identificador IPNS puede cambiar la vinculación. El registro, contenido en la DHT se propaga luego a través de la red IPFS mediante Kademlia. Cuando otros nodos quieren acceder al contenido, pueden buscar el registro usando el identificador IPNS y obtener el CID correspondiente.

IPNS es de interés particular para este proyecto debido a que permite crear enlaces persistentes a contenido que puede cambiar con el tiempo. Tal y como sucede con una URL que enlaza a un contenido compartido en plataformas de almacenamiento en la nube como Google Drive o Dropbox.

Este es el último apartado en esta explicación de los fundamentos de IPFS. Por supuesto, no se ha cubierto todo lo que IPFS ofrece, pero sí los conceptos fundamentales necesarios para entender el sistema que se propone en el proyecto.

2.3. Ecosistema en torno a IPFS

2.3.1. Introducción

El ecosistema IPFS es un conjunto diverso y creciente de proyectos, herramientas, comunidades e integraciones que trabajan colectivamente para desarrollar, adoptar y evolucionar IPFS. Este ecosistema es fundamental para el éxito y la adopción generalizada de IPFS, ya que proporciona una variedad de recursos y oportunidades para interactuar y construir sobre el mismo.

2.3.2. Proyectos basados en IPFS

Existen numerosos proyectos y productos que se basan en IPFS para ofrecer soluciones innovadoras y disruptivas en diferentes áreas de aplicación. Algunas de estas áreas son:

- Almacenamiento: proyectos que utilizan IPFS para proporcionar servicios de almacenamiento distribuido, persistente y rentable, como Filecoin, Sia, Storj o Textile.
- Alojamiento web: proyectos que utilizan IPFS para alojar sitios web estáticos o dinámicos sin depender de servidores centralizados, como Fleek, Pinata o Unstoppable Domains.
- Distribución de contenido: proyectos que utilizan IPFS para distribuir contenido multimedia, educativo o informativo de forma eficiente y descentralizada, como Audius, Wikipedia Mirror o Origin Protocol.
- Aplicaciones descentralizadas (dApps): proyectos que utilizan IPFS para construir aplicaciones web que funcionan sobre redes peer-to-peer, sin intermediarios ni puntos de fallo, como Brave, Metamask o OpenBazaar.

2.3.3. Herramientas y librerías de IPFS

Existen diversas herramientas y librerías que facilitan el desarrollo y la integración con IPFS, tanto para usuarios finales como para desarrolladores. Algunas de estas

herramientas y librerías son:

- Clientes API: herramientas que permiten interactuar con un nodo IPFS a través de una interfaz de programación de aplicaciones (API), como ipfs-http-client, ipfs-js or py-ipfs-http-client.
- Interfaces de línea de comandos: herramientas que permiten interactuar con un nodo IPFS a través de una terminal o consola, como ipfs or ipfs-cluster.
- Marcos de desarrollo: herramientas que facilitan la creación y el despliegue de aplicaciones basadas en IPFS, como Fleek, Textile or 3Box.
- Librerías para integrar IPFS: herramientas que permiten integrar IPFS en otras aplicaciones o plataformas, como ipfs-embed, js-ipfs or go-ipfs.

2.3.4. Comunidades en torno a IPFS

Existen diversas comunidades que se forman alrededor de IPFS, tanto para apoyar el desarrollo y la adopción del proyecto, como para explorar sus posibilidades y beneficios. Algunas de estas comunidades son:

- Comunidades de desarrolladores: comunidades que se dedican a contribuir al código fuente, a reportar errores, a proponer mejoras o a crear nuevas funcionalidades para IPFS, como el equipo principal de IPFS, los colaboradores externos o los grupos locales.
- Comunidades de usuarios: comunidades que se dedican a utilizar IPFS para sus propios fines, a compartir experiencias, a resolver dudas o a dar feedback sobre el proyecto, como los usuarios finales, los creadores de contenido o los operadores de nodos.
- Comunidades de gobernancia: comunidades que se dedican a definir las reglas, los principios y los objetivos del proyecto IPFS, así como a coordinar las acciones y los recursos necesarios para su cumplimiento, como el Protocol Labs, la Fundación Filecoin o el Consejo Asesor.

2.3.5. Integraciones de IPFS

Existen diversas formas en las que IPFS se integra con otras tecnologías y plataformas para ampliar sus capacidades y su alcance. Algunas de estas integraciones son:

- Ethereum: una plataforma de computación descentralizada que permite la creación de contratos inteligentes y aplicaciones descentralizadas. IPFS se integra con Ethereum para almacenar y distribuir los datos asociados a estas aplicaciones, así como para mejorar la escalabilidad y la eficiencia de la red.
- Filecoin: una red de almacenamiento descentralizado que permite a los usuarios alquilar o proveer espacio de disco a cambio de una criptomoneda. IPFS se integra con Filecoin para ofrecer una capa de incentivos económicos y una garantía de disponibilidad y persistencia de los datos almacenados en IPFS.

Como se puede apreciar IPFS es un proyecto con una gran comunidad y un ecosistema muy activo. Dada su naturaleza IPFS es un proyecto que se puede integrar en cantidad de ámbitos y tecnologías, ofreciendo las ventajas ya previamente mencionadas.

Capítulo 3

Estado del arte

En este capítulo se lleva a cabo un breve análisis de algunas implementaciones y plataformas que usan IPFS o tecnologías similares, así como de otras alternativas a IPFS, que cumplen parcial o completamente con los objetivos del proyecto.

Como se ha visto en el apartado anterior existen una gran cantidad de herramientas y proyectos que usan IPFS como base para sus sistemas. Pese a esto en el contexto de archivos compartidos, solo existen algunos proyectos interesantes que merece la pena analizar:

3.1. Peergos

Peergos es con una plataforma web para subir archivos y compartirlos con otros usuarios. Cuenta con un sistema de almacenamiento y comunicación descentralizado y seguro que utiliza criptografía y la red IPFS. Como se puede observar en la figura 3.1 Peergos proporciona una interfaz web para guardar, compartir y editar archivos, fotos, vídeos, mensajes y otros datos de forma privada y sin intermediarios. Peergos garantiza que solo los usuarios autorizados puedan acceder a sus datos, y que nadie pueda espiar o censurar su actividad en línea. Es una plataforma de código abierto y se puede ejecutar en cualquier dispositivo compatible con Java [18].

Este proyecto se encuentra en fase de desarrollo, aunque es un producto completo que además ofrece planes de almacenamiento como si de un proveedor en la nube se tratara. Esto es posible ya que al ser IPFS una red global, cualquier nodo de la red puede mantener *pinneados* (disponibles) los bloques que se deseen, y por tanto ofrecer un servicio de almacenamiento. En este caso Peergos ofrece un servicio de almacenamiento de pago, es decir, que tienen una serie de nodos en la red que mantienen pinneados los bloques de datos de los usuarios que pagan por el servicio.

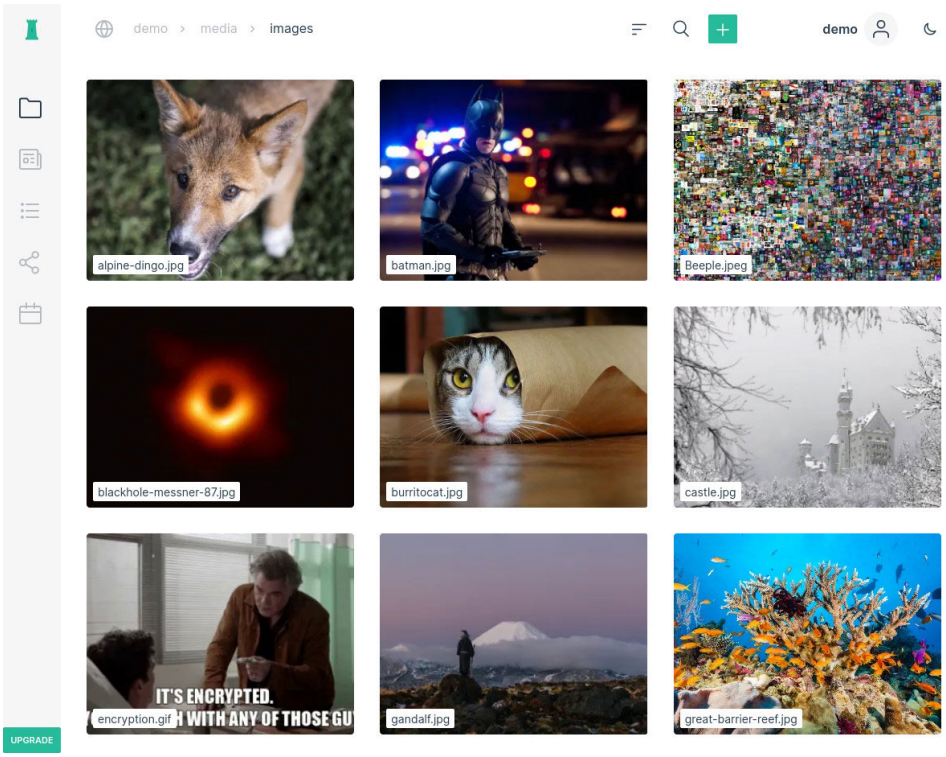

Figura 3.1: Plataforma web de Peergos

Otro aspecto de gran interés es el sistema registro de usuarios. Las claves públicas y los nombres de usuario se almacenan en una estructura de datos global de solo adición, con los nombres asignados por orden de llegada. Esto requiere consenso para garantizar la singularidad de los nombres de usuario. Aquí es también donde se almacena el ID del nodo IPFS del servidor (o servidores) responsables de sincronizar las escrituras del usuario. El problema de la implementación realizada por Peergos es que se necesita de uno o varios servidores centralizado que denominan *Corenode* que mantienen y sirven este registro para los usuarios.

Este registro es en sí la base de datos de usuarios, la autenticación se maneja mediante un sistema de clave pública-privada guardada en el navegador del usuario. Se añade una contraseña elegida por el usuario para acceder a la cuenta.

Peergos es una plataforma muy completa y que ofrece una gran cantidad de funcionalidades que escapan del alcance de este proyecto. Se puede considerar esta plataforma como referencia del potencial de IPFS, y aunque no es perfecta y tiene sus limitaciones, es un buen ejemplo de lo que se puede lograr con esta tecnología.

3.2. Filecoin

Filecoin[19] es una plataforma de almacenamiento en la nube descentralizada basada en blockchain. La plataforma utiliza su propia criptomoneda,llamada Filecoin (FIL), para facilitar e incentivar las transacciones dentro de la red.

Filecoin no está dirigido a consumidores (usuarios de a pie) ya que realmente es un mercado para proveedores de almacenamiento en la nube. Puede llegar a ser

de interés para el futuro de este proyecto ya que se podría integrar el sistema con Filecoin para ofrecer un servicio parecido a un proveedor de almacenamiento en la nube, pero con distintos proveedores que compiten entre sí para ofrecer el mejor servicio.

3.3. Sailplane

Sailplane se describe como una plataforma para *'Compartir archivos de forma colaborativa y punto a punto en el navegador'*.

Sailplane, al igual que este proyecto, usa OrbitDB como el componente central de sus sistema[1]. Como ya se ha explicado previamente, OrbitDB es una base de datos distribuida. Sailplane implementa un backend de almacenamiento conocido como un *store*. Un store en OrbitDB se refiere a una instancia de una base de datos individual que se sincroniza automáticamente con otros stores del mismo tipo mediante la red IPFS.

Sailplane ha implementado su propio store llamado *orbit-db-fsstore*[20]. Este representa un sistema de ficheros montado sobre OrbitDB que se puede sincronizar con otras instancias de la base de datos, lo que permite mantener y compartir un sistema de ficheros sincronizado entre varios usuarios. Este sistema de archivos se puede encriptar aunque no es necesario.

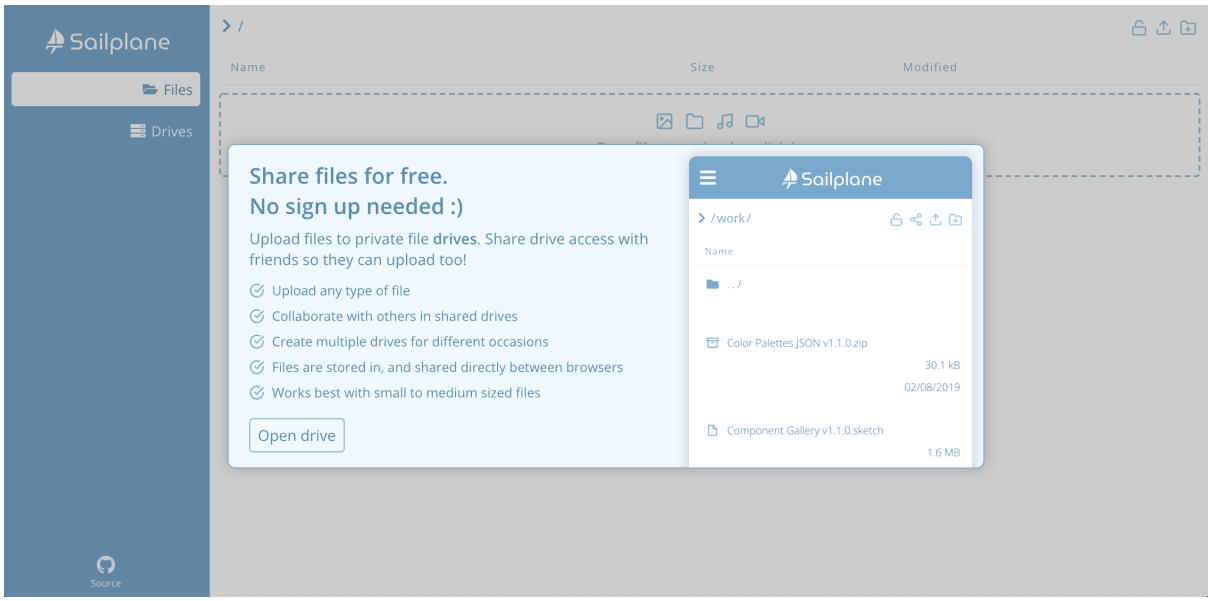

Figura 3.2: Aplicación web de Sailplane

Sailplane ofrece una implementación de un nodo IPFS personalizado llamado *sailplane-node*[21], que expone una interfaz para interactuar directamente con el store. Existe también una web que hace uso de este nodo y ofrece:

- Un sistema de ficheros en el navegador basado en *drives* (discos virtuales) com-

[1]Este proyecto fue descubierto hacia el final del desarrollo de este trabajo de fin de grado por lo que este hecho es más bien una casualidad.

partidos.

- Un sistema de registro automático y autocontenido. Esto sirve para que la hora de compartir un drive este pueda ver los usuarios con los que puede compartirlo.

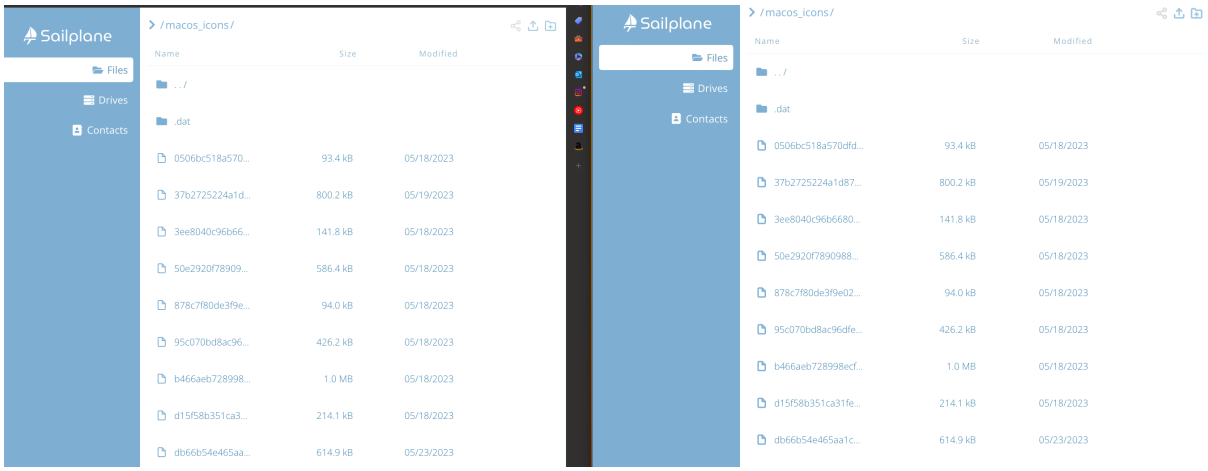

Figura 3.3: Dos nodos Sailplane sincronizando el el mismo drive

Lo que más destaca de Sailplane es la implementación de un sistema de ficheros montado sobre OrbitDB. Esto es algo que se explorará más adelante en la el capítulo 7: 'Trabajos futuros'. El uso de OrbitDB para un sistema de registro automático y autocontenido es algo que también se ha implementado en la propuesta, aunque como se ha comentado previamente, el descubrimiento de este proyecto ocurrió ya habiendo desarrollado esta característica.

3.4. Fileverse

Fileverse es una plataforma de almacenamiento que se integra con IPFS y que permite guardar, compartir y acceder a archivos desde cualquier dispositivo.

Actualmente ofrece dos productos, ambos en formato de aplicación web:

- Fileverse Solo: Una aplicación web para subir y compartir archivos sin autenticación. El usuario sube un archivo y recibe un enlace que puede compartir con otros usuarios. En la figura 3.4 se puede observar un ejemplo de uso.
- Fileverse Portals: Una aplicación web que integra autenticación basada en blockchain. Es un espacio de trabajo para la gestión de archivos sobre blochain (e IPFS) y la creación de contenido. Está en fase de pruebas y no se puede acceder.

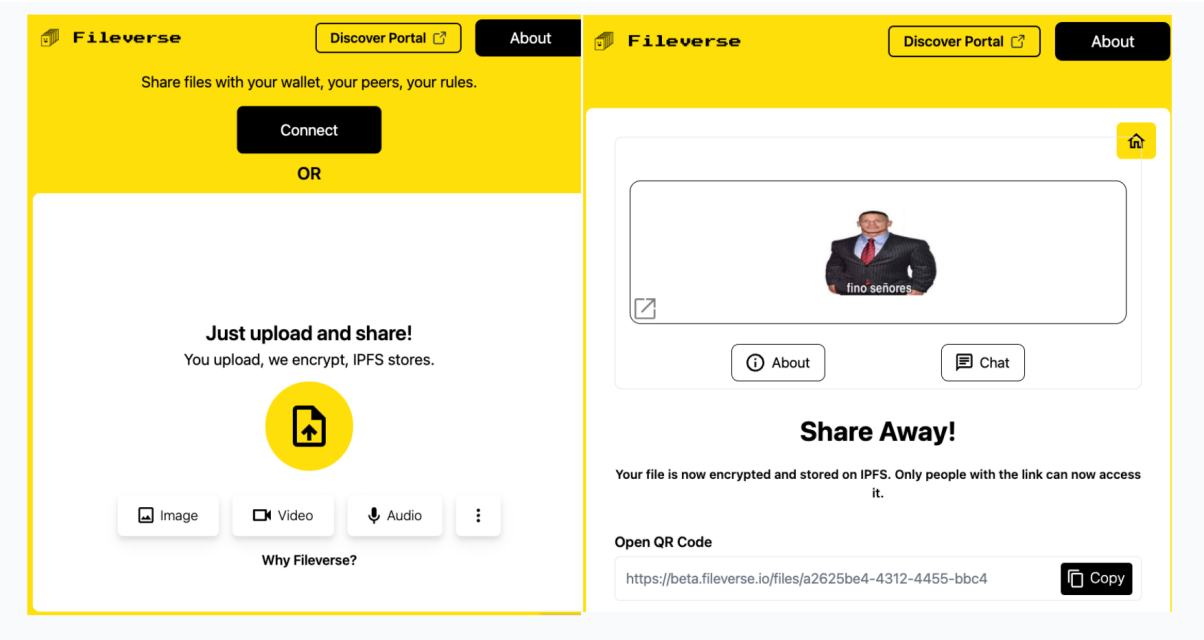

Figura 3.4: Aplicación web de Fileverse para subir archivos

El servicio que ofrece Fileverse Solo es muy simple: el usuario sube un archivo y recibe un enlace que puede pasar a otras personas para descargar dicho archivo. Esta caso de uso es muy similar al que se propone en este proyecto, aunque con algunas diferencias. El uso de una aplicación web resulta más sencillo que una línea de comandos, pero no integra cuentas de usuario ni formas de acceder o gestionar comparticiones pasadas.

3.5. Conclusión

Como se puede observar en este capítulo existen varias implementaciones de sistemas de almacenamiento y compartición de archivos basadas en IPFS. Como opinión personal, ninguna presenta un servicio comparable al de los grandes proveedores centralizados de almacenamiento en la nube. Tanto en términos de facilidad de uso, como de funcionalidades y fiabilidad.

Esto se debe a que detrás de estos servicios hay una gran infraestructura e inversión que no es comparable a estos proyectos que tienen un carácter más experimental, investigativo y proviene de equipos de desarrollo mucho más limitados. Aún así, estos proyectos son un buen ejemplo de lo que se puede lograr con IPFS y han sido de gran ayuda a la hora de realizar este proyecto, en particular Sailplane y Peergos.

Capítulo 4

Desarrollo de IPFShare

Se ha denominado IPFShare al sistema de intercambio de archivos desarrollado en este proyecto.

4.1. Requisitos y definición del sistema

En esta sección se definen los requisitos funcionales y no funcionales que debe integrar el sistema propuesto.

4.1.1. Requisitos funcionales

- Un usuario debe poder autenticarse en el sistema.
- Un usuario debe poder compartir archivos y directorios con uno o varios usuarios.
- Un usuario debe poder descargar archivos y directorios compartidos por otros usuarios del sistema.
- A la hora de compartir un archivo un usuario debe poder elegir con qué otros usuarios del sistema compartirlo.
- Un usuario debe tener posibilidad de hacer grupos de compartición.
- Cuando un usuario comparta recursos con otros usuarios, estos deben recibir una notificación.
- Un usuario debe poder interactuar las comparticiones que ha realizado.
- Un usuario debe poder interactuar las comparticiones que le han realizado.

4.1.2. Requisitos no funcionales

- El sistema debe proporcionar una interfaz de usuario intuitiva y fácil de usar.
- El sistema debe implementar servicios de seguridad en torno a la autenticación.
- El sistema debe proporcionar medidas de seguridad para los archivos compartidos.
- El sistema debe proporcionar un servicio o mecanismos de identificación de usuario portable.
- Las transacciones y eventos del sistema deben ser relativamente instantáneos.

Hay que destacar que este sistema no propone la sincronización de archivos que

normalmente se asocia con los sistemas de almacenamiento en la nube. Incluir el desarrollo de esta funcionalidad en el sistema propuesto no es algo trivial. La sincronización de datos en tiempo real es un tópico muy complejo y se sale del alcance de este proyecto. Sin embargo, en la sección de Trabajos futuros dentro de las posibles mejoras se propone una posible implementación de esta funcionalidad.

4.2. Arquitectura y diseño

Uno de los objetivos de este trabajo es comparar la viabilidad de IPFS como alternativa a los sistemas centralizados. En esta sección se presenta una arquitectura simplificada para un sistema de intercambio de archivos centralizado habitual. Después se contrastará con la alternativa descentralizada propuesta.

4.2.1. Arquitectura de un sistema de intercambio de archivos centralizado habitual

Como se puede observar en la figura 4.1, la arquitectura de un sistema de intercambio de archivos requiere de varios puntos de acceso centralizados para funcionar[1].En un caso real esta arquitectura podría ser más compleja, implementando sistemas de caching, entre otras muchas mejoras posibles. Para este ejemplo se ha simplificado para centrarse en los puntos clave.

Existe una clara separación conceptual de las distintos componentes, aunque pueden estar ubicados en el mismo servidor físico, en varios, o incluso, cada servicio en varios servidores físicos. Esto depende de la escala del sistema y de las necesidades de rendimiento y disponibilidad.

- **Almacenamiento cloud:** el almacén donde se guardarán los chunks de ficheros proporcionados por el servicio de procesamiento de ficheros, Amazon S3 es un ejemplo de un servicio de estas característica.
- **Servicio de procesamiento de ficheros:** es el encargado atender y llevar a cabo las peticiones de usuario relacionadas con ficheros.
- **Servicio de autenticación:** gestiona la autenticación de los usuarios. Contiene los datos de los usuarios y sus credenciales.
- **Servicio de metadatos:** se encarga de enviar las notificaciones de los usuarios. Cuando un cliente comparte un fichero con otro, el cliente envía también una serie de metadatos sobre esta acción que acaba de realizar. Estos metadatos se pueden usar para gran variedad de propósitos. En este caso se usan para implementar un servicio de notificaciones basado en colas de mensajes.

Cuando un cliente quiere compartir un fichero, este divide el archivo en trozos más pequeños llamados chunks[2]. Envía a través de una conexión segura (socket tpc sobre TLS por ejemplo) estos chunks que componen el archivo al servicio de procesamiento de ficheros. Este servicio se encarga de guardar los chunks en el almacenamiento cloud y de mandar los metadatos a la cola de mensajes. El servicio de metadatos

[1]Este diseño se ha ideado tomando como referencia los siguientes recursos: [22] y [23].

[2]En un servicio de compartición de archivos, los archivos se subdividen en fragmentos o chunks para mejorar la eficiencia de la transferencia de datos, permitiendo la transmisión paralela y la reanudación de las transferencias interrumpidas, así como facilitar la distribución del almacenamiento y reducir la redundancia de datos.

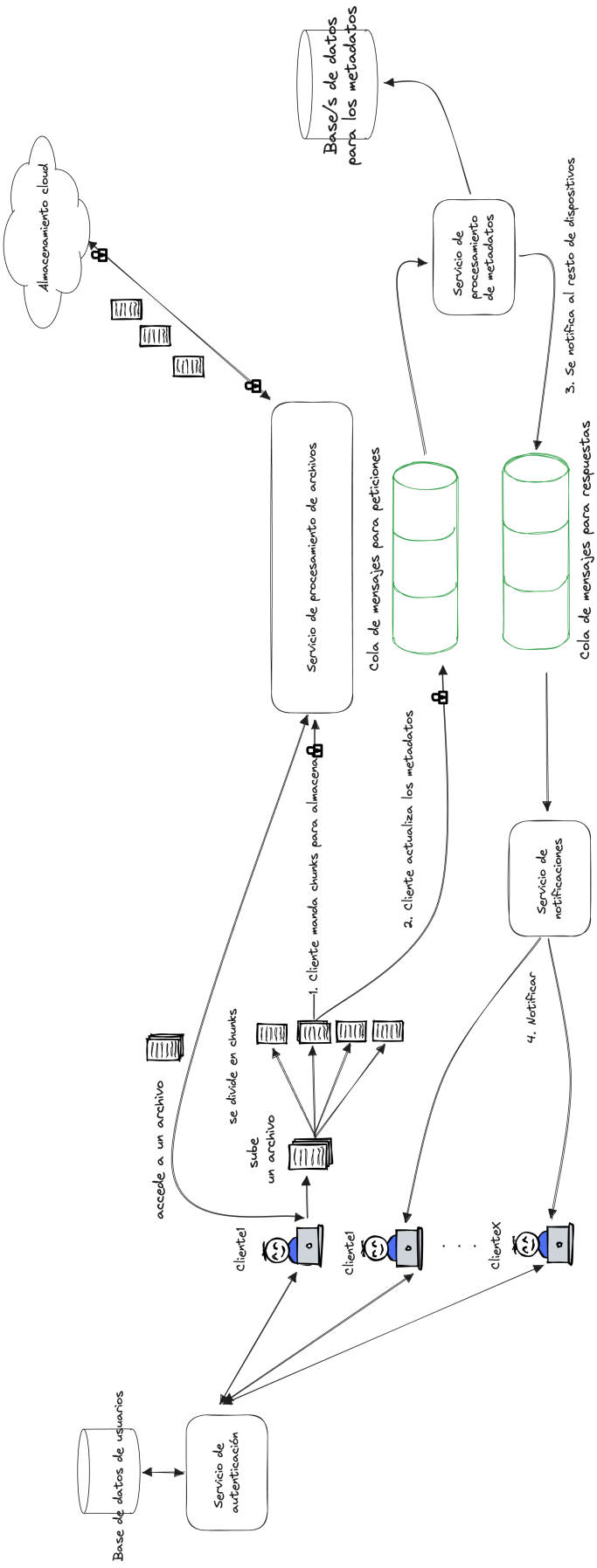

Figura 4.1: Posible arquitectura de un servicio centralizado de archivos

procesa de forma asíncrona estos metadatos, extrayendo la información pertinente que identifique a los usuarios y el recurso compartido. Después envía una notificación a los usuarios que corresponda.

Dado que el servicio de autenticación posee datos de los usuarios, cuando un usuario quiera enviar un archivo puede pedir esta al servicio de autenticación, el incluso se podría combinar con el servicio de metadatos para brindar otras funcionalidades como ver los recursos que se le han compartido o los usuarios con los que se ha compartido.

Bajo este sistema los clientes deben saber de antemano dónde se ubican estos servicios para poder interactuar con ellos. Estos servicios podrían ser provistos por entidades externas (lo cual es muy habitual en un sistema de este estilo), creando dependencias en sistemas centralizados de terceros.

4.2.2. Arquitectura de IPFShare: un sistema de intercambio de archivos descentralizado

En comparación con la arquitectura anterior (figura 4.1), la arquitectura de IPFShare es completamente descentralizada. No existe ningún servicio centralizado, cualquier servicio proporcionado por una entidad externa proviene de de otros nodos IPFS, siendo el propio nodo cliente otro proveedor más de servicios. Esto hace el sistema completamente autocontenido e independiente de servicios fuera de la red IPFS.

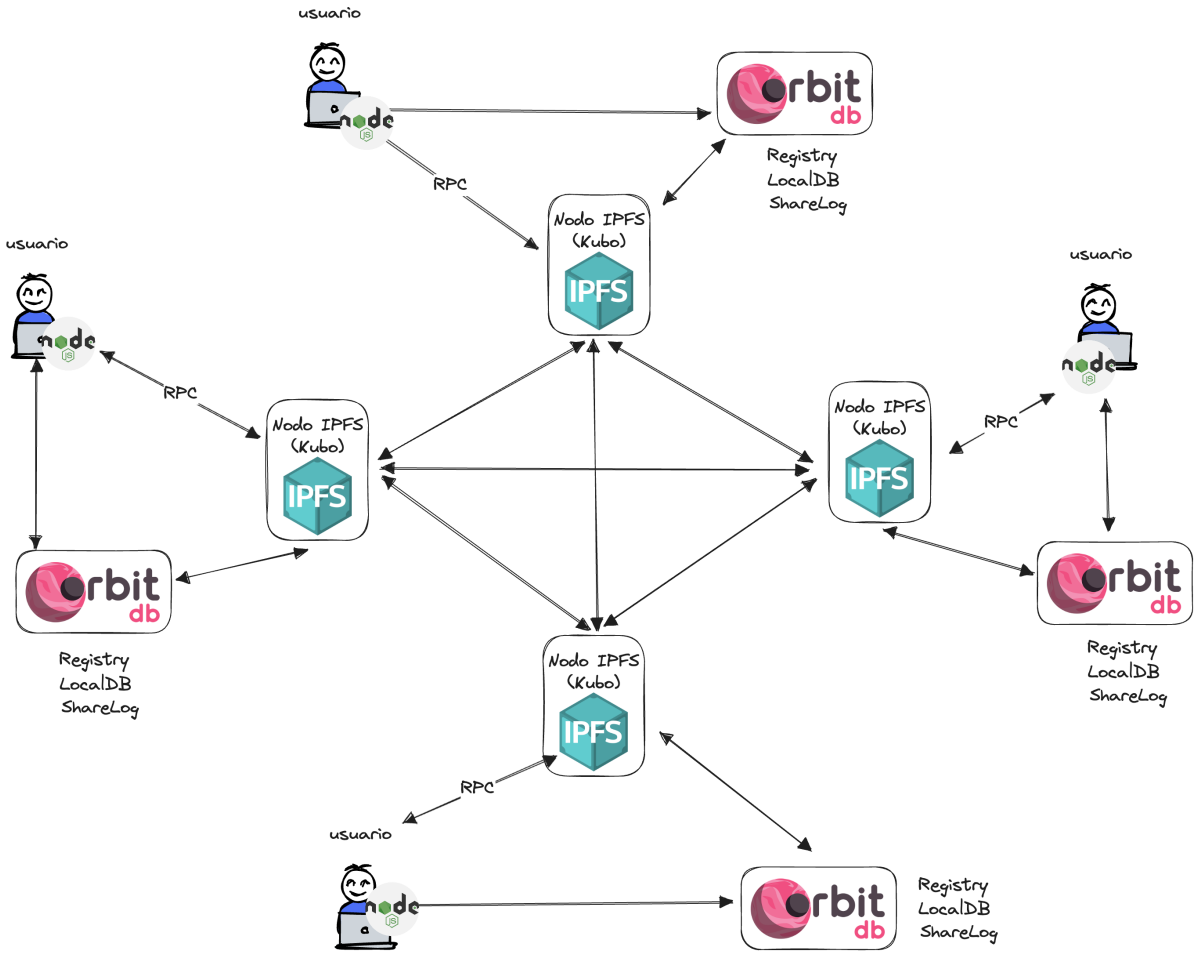

Figura 4.2: Arquitectura de IPFShare

En el sistema propuesto, cada usuario opera un nodo IPFS, que sirve como su punto de conexión a la red IPFS. Las interacciones entre el cliente y este nodo IPFS se facilitan a través de una interfaz de programación de aplicaciones (API), que expone un conjunto de operaciones que puede realizar el nodo [24]. Esta comunicación se lleva a cabo mediante RPC, un protocolo que permite al cliente ejecutar procedimientos en el nodo IPFS como si estuvieran en el mismo sistema, lo que facilita la solicitud eficiente de servicios y operaciones.

El programa interactúa con la red IPFS mediante esta API a través del nodo IPFS local. IPFS sustituye al servicio de almacenamiento cloud al ser un gran swarm de nodos que almacenan y comparten datos entre sí. IPFS en sí mismo no impone un límite teórico sobre la cantidad de datos que pueden ser subidos a un nodo que se comporta correctamente. El límite práctico vendría determinado por factores como los recursos del sistema (espacio en disco, ancho de banda de red, etc.), la configuración del nodo IPFS y cualquier restricción impuesta por la red o el proveedor de alojamiento. Proveedor de alojamiento quiere decir otros nodos IPFS que compartan el mismo (o parte del) pinset que el nodo del usuario.

El servicio de procesamiento de ficheros se sustituye también por IPFS. El proceso de añadir cualquier dato a IPFS implica la subdivisión en chunks para la generación del

DAG y el uso de este en Bitswap, como ya se explicó en el apartado 2.2.4: 'Modelo de datos'.

4.2.3. Autenticación y control de acceso

El servicio de autenticación se sustituye por un sistema de identificación descentralizado basado en DIDs que opera mediante lo que se ha denominado como un *Registry*. Un Registry es una base de datos distribuida que almacena información sobre los usuarios del sistema. Esta base de datos distribuida es provista por OrbitDB, del cual se hablará más tarde en el apartado 4.3.1.

El concepto principal consiste en que cada cliente tiene un instancia de OrbitDB que opera mediante IPFS. Esta instancia de OrbitDB mantiene constancia de una serie de bases de datos distribuidas que se sincronizan con el resto de nodos, intercambiando los datos que se han modificado o añadido. De esta forma, se consigue una base de datos replicada y consistente entre todos los participantes del sistema. OrbitDB ofrece diferentes tipos de bases de datos, como colecciones de documentos, claves-valor o registros de eventos. Para el caso del Registry, se utiliza una base de datos de tipo clave-valor, donde la clave es el DID del usuario y el valor es un objeto JSON que contiene el peerID del usuario, su DID y un alias.

- El DID es un identificador único que se genera a partir de la clave privada del usuario. Se utiliza como identidad de la instancia de OrbitDB para permitir verificar la autoría de las entradas en el registro.
- El alias es un nombre que el usuario puede elegir para identificarse en el sistema. Puede hacerse único o no, dependiendo de la política de acceso del Registry.

Un Registry debe incorporar un controlador de acceso que mantiene una serie de políticas de acceso respecto del mismo. Estas políticas son:

- Cualquier usuario puede crear una entrada en el registro.
- Un usuario solo puede tener una entrada en el registro.
- Solo el usuario que creó la entrada una puede modificarla.
- Solo el usuario que creó la entrada una puede eliminarla.

Cuando se crea una instancia de OrbitDB, el creador especifica un controlador de acceso (AC) para la base de datos. Este controlador de acceso se almacena en el manifiesto de la base de datos (los metadatos de la base de datos) y la dirección de este manifiesto se utiliza luego para cargar la base de datos. Entonces surge la pregunta, ¿no podría alguien simplemente intercambiar el controlador de acceso en su instancia local?
La respuesta a esto es no, no pueden por las siguiente razones:

1. El controlador de acceso y sus parámetros se almacenan en el manifiesto de la base de datos, y el hash de este manifiesto compone la dirección de la base de datos. Si alguien cambia el controlador de acceso, cambiarían el manifiesto, y por lo tanto la dirección de la base de datos. Esencialmente, estarían creando una base de datos completamente nueva, no alterando la original.
2. Incluso si alguien crea una nueva base de datos con un controlador de acceso diferente, no pueden escribir en la base de datos original al no concordar las

direcciones.
3. Todas las actualizaciones a la base de datos (adiciones, modificaciones, etc.) son firmadas por las identidad del escritor. Si un usuario malintencionado intenta alterar los datos, las firmas no coincidirán, haciendo que los cambios sean invalidados por el resto de nodos que poseen la base datos.
4. Al leer de la base de datos, el controlador de acceso valida si los datos (entradas) pueden ser añadidos a la base de datos local o no. Incluso si alguien cambiara el controlador de acceso localmente, no podrían alimentar datos alterados a otros porque los controladores de acceso de los demás validarían y rechazarían los cambios.

4.2.4. Colas de mensajes asíncronas distribuidas

El servicio de metadatos mediante colas de mensajes se sustituye por una base de datos distribuida de tipo log, también provista por OrbitDB. Este componente se ha denominado en el sistema *ShareLog*. Un ShareLog es una base de datos distribuida que mantiene un registro de las acciones de compartición de los usuarios. Al realizar una compartición, el cliente añade una entrada al ShareLog, que contiene información sobre esta, que es distribuida entre todos los nodos del sistema. De esta forma, se consigue un registro de las comparticiones que se ha realizado en el sistema, que se puede consultar en cualquier momento. En este sistema cuando se habla de notificaciones se refiere a entradas que llegan del ShareLog a los nodos clientes, que las procesan y muestran notificaciones al usuario, si es que están dirigidos a este.

4.2.5. Protocolo de compartición

El diagrama siguiente muestra como funciona la compartición en IPFShare:

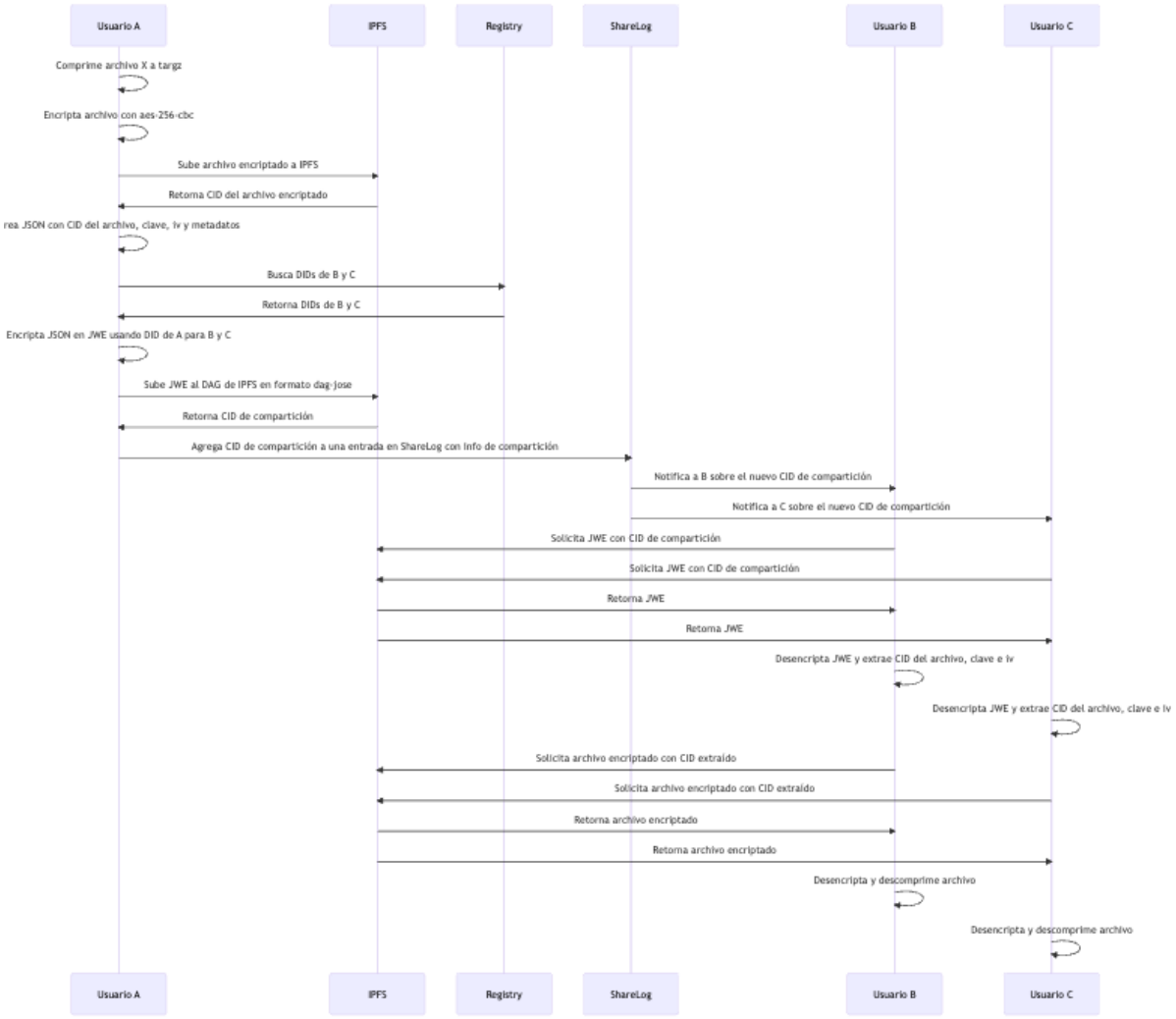

Figura 4.3: Protocolo de compartición implementado por IPFShare

En palabras este protocolo funciona de la siguiente forma:

- Un archivo designado como “X” es comprimido en formato ’targz’ por el usuario A.
- Posteriormente, el archivo comprimido es encriptado por el usuario A mediante el algoritmo aes-256-cbc.
- El archivo encriptado es subido al sistema Interplanetary File System (IPFS) por el usuario A. Como resultado de este proceso, el sistema IPFS devuelve un identificador de contenido (CID) correspondiente al archivo encriptado.
- El usuario A procede a la creación de un JSON que incluye el CID del archivo encriptado, la clave utilizada para la encriptación, el vector de inicialización (iv) y metadatos adicionales.
- El usuario A consulta el Registro (Registry) para obtener los Identificadores Descentralizados (DID) correspondientes a los usuarios B y C.
- Utilizando los DIDs obtenidos, el usuario A encripta el JSON previamente creado en un Token de Seguridad de JSON Encriptado (JWE). Esta encriptación se realiza de manera que únicamente los usuarios B y C pueden leer el contenido.

- El JWE es subido al Gráfico Acíclico Dirigido (DAG) de IPFS por el usuario A en formato dag-jose. En respuesta a esta operación, IPFS devuelve un CID correspondiente al JWE.
- Este CID, referido como el CID de compartición, es añadido por el usuario A a una entrada en el ShareLog con información detallada acerca de la compartición.
- ShareLog procede a notificar a los usuarios B y C sobre el nuevo CID de compartición.
- Los usuarios B y C solicitan a IPFS el JWE utilizando el CID de compartición. IPFS retorna el JWE a los usuarios.
- Posteriormente, los usuarios B y C desencriptan el JWE, y extraen de él el CID del archivo encriptado, la clave y el iv.
- Utilizando el CID extraído, los usuarios B y C solicitan a IPFS el archivo encriptado.
- Una vez obtenido, los usuarios B y C desencriptan y descomprimen el archivo utilizando la clave y el iv extraídos.

4.3. Implementación

Esta implementación se encuentra disponible en Github bajo la licencia MIT.

4.3.1. Tecnologías usadas

- **NodeJS:** Node.js es un entorno de ejecución (runtime) de JavaScript de código abierto, multiplataforma, que ejecuta código JavaScript fuera de un navegador web. NodeJS tiene una gran comunidad y dispone de una gran variedad de paquetes para uso dentro de su ecosistema. En este proyecto se usa NodeJS como runtime sobre el que se ejecuta la aplicación de línea de comandos IPFShare.
- **Typescript:** Typescript es un lenguaje de programación de código abierto desarrollado y mantenido por Microsoft. Es un superconjunto de JavaScript, que esencialmente añade tipos estáticos a Javascript. Se ha decidido usar este lenguaje en vez de JavaScript debido a que el tipado estático ayuda a detectar errores en tiempo de compilación, lo que facilita el desarrollo y reduce la cantidad de errores en tiempo de ejecución. Typescript realmente es transpilado a Javascriopt, esto permite usar paquetes de tipo ESM y CommonJS por igual sin tener que preocuparse por la compatibilidad entre los dos sistemas de módulos.
- **Paquetes/módulos de NodeJS**:
  - `archiver`: Proporciona funcionalidades para generar archivos comprimidos (por ejemplo, zip o tar) y trabajar con ellos.
  - `chalk`: Herramienta para formatear y colorear el texto de la consola de Node.js.
  - `cli-progress`: Permite crear barras de progreso en la interfaz de línea de comandos.
  - `clipboardy`: Facilita el acceso al portapapeles del sistema para copiar y pegar.
  - `commander`: Proporciona utilidades para escribir scripts de línea de comandos en Node.js.
  - `did`: Biblioteca para trabajar con Identificadores Descentralizados (DIDs).
  - `did-jwt`: Proporciona métodos para crear y verificar JWTs que usan DIDs.

- `did-resolver`: Ofrece funcionalidades para resolver DIDs a través de varios métodos.
- `dids`: Otra biblioteca para trabajar con DIDs, proporciona una API sencilla para crear y gestionar DIDs.
- `figlet`: Permite crear banners de texto ASCII.
- `fs-extra`: Proporciona métodos adicionales para trabajar con el sistema de archivos que no se incluyen en el módulo `fs` de Node.js.
- `go-ipfs`: Descarga el binario de IPFS, Kubo, en su útlima versión para su uso en cada plataforma.
- `inquirer`: Biblioteca para crear interfaces de usuario interactivas en la línea de comandos.
- `ipfsd-ctl`: Biblioteca para controlar un demonio IPFS desde Node.js.
- `key-did-provider-ed25519`: Proporciona un proveedor DID basado en el esquema de firma Ed25519.
- `key-did-resolver`: Ofrece métodos para la resolución de DIDs.
- `kubo-rpc-client`: Un cliente RPC para Kubo.
- `node-notifier`: Permite enviar notificaciones push nativas a cada plataforma dentro de Node.js.
- `ora`: Proporciona spinners elegantes para usar en la interfaz de línea de comandos.
- `orbit-db`: Base de datos peer-to-peer, construida sobre IPFS.
- `orbit-db-access-controllers`: Proporciona controladores de acceso para OrbitDB.
- `orbit-db-identity-provider`: Un proveedor de identidad para OrbitDB. Este paquete ha tenido que ser modificado para funcionar correctamente con DIDs como identidades debido a un error en su implementación. Disponible en Github.
- `orbit-db-keystore`: Almacén de claves para OrbitDB.
- `prompts`: Biblioteca para crear diálogos de usuario interactivos en la línea de comandos.
- `ps-list`: Ofrece una forma de enumerar todos los procesos en ejecución en el sistema.
- `tar`: Proporciona funcionalidades para trabajar con archivos tar.
- `winston`: Un logger flexible para Node.js.

Se ha decidido usar la implementación de IPFS escrita en Go, llamada Kubo debido a que los problemas existentes con la implementación de IPFS escrita en Javascript llamada js-ipfs. Al usar este proyecto NodeJS sería óptimo el uso de la implementación de IPFS escrita en Javascript al ser una solución más integrada. Sin embargo, a la hora de hacer pruebas js-ipfs sufre de muchos problemas de conectividad a través del mecanismo de pubsub necesario para el uso de OrbitDB.

Kubo, por otro lado, es una implementación mucho más estable y madura, que no sufre de estos problemas, lo que hace que sea más adecuada para el uso en producción. La API RPC que integra y que es accesible desde NodeJS tiene una integración suficiente para el uso de IPFShare. El único problema que se ha encontrado es que el uso de Kubo desde NodeJS no permite usar una instancia personalizada de libp2p, esta instancia permite un control más granular sobre las comunicaciones y añade funcionalidades que no están disponibles en la API RPC.

4.3.2. Línea de comandos: CLI

Se ha desarrollado una interfaz de línea de comandos (CLI) para facilitar el uso del sistema. Esta CLI se ha desarrollado usando CommanderJS. Para crear una CLI en CommanderJS se sigue el siguiente patrón:

- Se crea una instancia de Command.
- Se añaden comandos, opciones y argumentos a esta instancia usando una serie de métodos proporcionados.
- Se puede definir la acción a realizar al ejecutar cada comando.
- Se llama al método `parse` o *parseAsync* para que CommanderJS procese los argumentos de la línea de comandos y ejecute las acciones correspondientes.

Por ejemplo:

```
import { Argument, Command, Option, OptionValues } from "@commander-js/extra-typings"
const program = new Command()

program
    .version("0.0.1")
    .name("ipfshare")
    .addHelpText("before", `${chalk.yellow(logo)}`)
    .addHelpText("before", "An IPFS-based, encrypted file sharing CLI tool\n")
    .action(async () => {
        await notSetupPrompt()
        await daemonPromptIfNeeded() // checks if the daemon is running, if not it will prompt the user to start it
        program.help()
    })


    program.command("setup")
    .summary("Run initial setup")
    .description(`Runs the initial setup:
    - Creates IPFShare home folder. This is where all files/folders program related are located
    - Generate the IPFS repository and config
    - Generates encryption keys
    - Etc.`
    )
    .argument("[path]", "Path to IPFShare home folder") // Square brackets around the argument make it optional
    .action(async (path) => {
        if (path) {
            return await setupPrompt(path)
        }
        await setupPrompt()
    })
```

Estos son los comandos implementados por IPFShare:

```
$ ipfshare --help
```

```
,,
`7MMF'`7MM"""Mq.`7MM"""YMM  .M"""bgd `7MM
MM    MM   `MM. MM    `7 ,MI    "Y   MM
MM    MM   ,M9  MM   d   `MMb.       MMpMMMb.   ,6"Yb.  `7Mb,od8 .gP"Ya
MM    MMmmdM9   MM""MM     `YMMNq.   MM    MM  8)   MM    MM' "',M'   Yb
MM    MM        MM   Y  .      `MM   MM    MM   ,pm9MM    MM    8M""""""
MM    MM        MM     Mb     dM   MM    MM  8M   MM    MM    YM.    ,
.JMML..JMML.    .JMML.    P"Ybmmd"  .JMML  JMML.`Moo9^Yo..JMML.   `Mbmmd'

An IPFS-based, encrypted file sharing CLI tool
```

```
Usage: ipfshare [options] [command]

Options:
-V, --version                   output the version number
-h, --help                      display help for command

Commands:
setup [path]                    Run initial setup
daemon <action>                 Start the Kubo (go-ipfs) daemon. This
 ↪  is a custom daemon for IPFShare. See daemon --help for more
info.
cat [cids...]                   Print the contents of a given CID
ls [cids...]                    List the contents of a given CID
share [path...]
download [options] [cids...]    Print the contents of a given CID
sharelog                        Interact with the global share log
shared                          List all files and folders shared with
 ↪  you
shares                          Interact with shares created by you
registry                        Access the IPFShare global registry.
 ↪  Change username or delete account.
info                            Provides information about the running
 ↪  ipfshare instance such as DID and peerID
```

Cuando se ejecuta un comando la aplicación realiza todas las operaciones correspondientes dentro de lo que se ha denominado como *Contexto*. El Contexto es un objeto que contiene todas las instancias de las clases que se necesitan para el correcto funcionamiento de la aplicación. Dado que existen interdependencias entre los componentes del Contexto este se tiene que iniciar en una secuencia específica.

```
1  import { AppConfig } from "@app/common/appConfig"
2  import { UserRegistry } from "@app/registry"
3  import { IPFShareLog } from "@app/shareLog.js"
4  import { IPFSNodeManager } from "@ipfs/IPFSNodeManager.js"
5  import { DID } from "dids"
6  import type { Controller, ControllerType } from "ipfsd-ctl"
7  import { Socket } from "net"
8  import OrbitDB from "orbit-db"
9  import { Identity } from "orbit-db-identity-provider"
10
11 export interface AppContext {
12   // fábrica de nodos IPFS preconfigurados
13   manager: IPFSNodeManager | undefined
14   orbitdb: OrbitDB | undefined // instancia de OrbitDB
15   dbAddress: string | undefined // dirección de la base de datos
16   identity: Identity | undefined // identidad de OrbitDB
17   // DID de la identidad (es el mismo que identity pero en formato DID específico)
18   did: DID | undefined
19   // el nodo IPFS o controlador
20   ipfs: Controller<ControllerType> | undefined
21   // socket para comunicarse con el proceso daemon para el protocolo de control de pausa y
     ↪  reanudación
22   daemonSocket: Socket | undefined
23   // el Registry global de IPFShare
24   registry: UserRegistry | undefined
25   // el ShareLog global de IPFShare
26   shareLog: IPFShareLog | undefined
27   // objeto con la configuración de la aplicación proviene de un archivo de la instalación
```

```
28    appConfig: AppConfig | undefined //
29  }
30  export async function withContext(fn: () => Promise<void>) {
31    await initializeContext()
32    await fn()
33    await deInitializeContext()
34  }
```

La función que se pasa como argumento a withContext se ejecuta dentro del contexto asegurando que este se ha inicializado correctamente, y que se desinicializa en el órden adecuado después de terminar la ejecución de la función.

Estas son las secuencias de inicialización y desinicialización del contexto:

```
1   export async function initializeContext() {
2     if (ctx.manager === undefined) {
3   // fábrica de nodos IPFS preconfigurados
4       ctx.manager = await new IPFSNodeManager()
5   // crear un controlador (API RPC) que se conecta al daemon IPFS
6       let ipfs : Controller<"go"> = await ctx.manager.createNode()
7       ipfs = await ipfs.start().catch((err) => {
8           logger.error(err)
9           throw err
10      })
11      ctx.ipfs = ipfs
12      ctx.daemonSocket = new net.Socket()
13      ctx.daemonSocket.connect(3000, "localhost", () => {
14          logger.info("Connected to the daemon process.")
15      })
16      await sendMessageToOrbitDBService("pauseOrbitDB")
17      ctx.orbitdb = await getOrbitDB()
18
19      ctx.registry = new UserRegistry(ctx.ipfs.api, ctx.orbitdb)
20      await ctx.registry.open()
21      ctx.shareLog = new IPFShareLog(ctx.ipfs.api, ctx.orbitdb, "ipfs-sharelog")
22      await ctx.shareLog.open()
23
24      ctx.appConfig = await getAppConfigAndPromptIfUsernameInvalid()
25    }
26  }
27  export async function deInitializeContext() {
28    await ctx.registry?.store.close()
29    await ctx.shareLog?.store.close()
30    if (!ctx.orbitdb) throw new Error("OrbitDB instance undefined, not closed yet, should not
      ↪  happen")
31    ctx.identity?.provider.keystore.close()
32    await ctx.orbitdb?.disconnect()
33    ctx.daemonSocket?.write("resumeOrbitDB")
34    process.exit(0)
35  }
```

4.3.3. Protocolo de control de pausa y reanudación entre demonio y CLI

Como se puede observar dentro del Contexto se implementa parte del protocolo de control de pausa u reanudación que comunica con el proceso demonio que se ejecuta en segundo plano. Este protocolo se ha implementado para poder usar dos instancias de OrbitDB bajo el mismo directorio de datos y así evitar problemas de concurrencia.

Esto es necesario ya que el demonio siempre está en ejecución, con su instancia de OrbitDB manteniendo las bases de datos actualizadas y sincronizadas con el resto de nodos. Cuando se ejecuta la CLI, esta instancia de OrbitDB se pausa para que la CLI pueda '*robar*' la instancia al demonio (la vuelve a crear). Cuando se termina la ejecución de la CLI, se reanuda la instancia de OrbitDB del demonio para que este pueda seguir funcionando, retransmitiendo las operaciones que haya realizado la CLI.

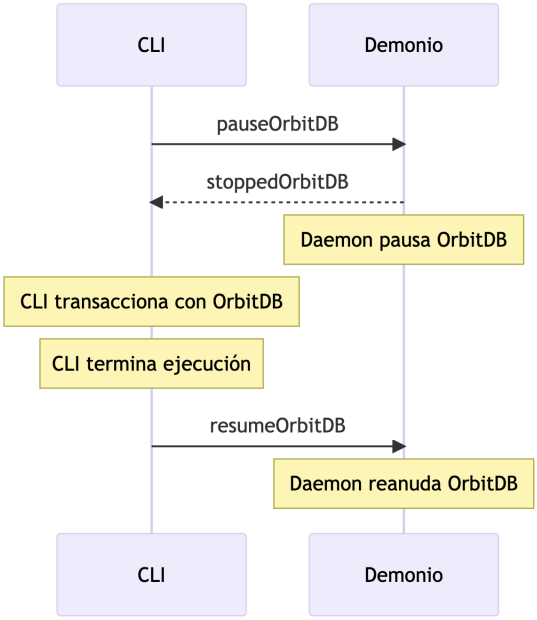

Figura 4.4: Diagrama de secuencia del protocolo de control de pausa y reanudación implementado

La comunicación se realiza mediante un socket TCP en el puerto 3000 (se podría configurar en cualquier otro puerto). El protocolo es muy sencillo: el cliente envía un mensaje al servidor, el servidor lo procesa y responde con otro mensaje de confirmación. El cliente espera a recibir este mensaje para continuar con la ejecución. Cuando termina manda otro mensaje al servidor para que este retome donde lo había dejado.

```
// esta es la función que se encarga de enviar comunicarse con proceso demonio
function sendMessageToOrbitDBService(message: string): Promise<string> {
  return new Promise((resolve, reject) => {
      if (ctx.daemonSocket === undefined ) throw new Error("Daemon socket is undefined")
      ctx.daemonSocket.once("data", data => {
          // Handle any response or acknowledgment from the daemon process
          const response = data.toString()
          logger.debug(`Received response for message ${message}:`, response)
          resolve(response)
      })

      ctx.daemonSocket.write(message, error => {
          if (error) {
              console.error("Error sending message:", error.message)
              reject(error)
          }
          logger.debug(`Sent message ${message}`)
      })
  })
}
// este servicio es creado por el proceso demonio y se encarga de procesar los mensajes que recibe de la CLI
private static createOrbitDBService() {
const server = net.createServer(socket => {
  socket.on("data", data => {
    const message = data.toString()

    if (message === "pauseOrbitDB") {
        (async () => {
            logger.info("Pausing OrbitDB")
            if (!ctx.orbitdb) throw new Error("OrbitDB is not initialized")
            await ctx.orbitdb.disconnect()
            if (!process.env.IPFSHARE_HOME) throw new Error("IPFShare home is not set")
```

```
            ctx.identity?.provider.keystore.close()
            socket.write("OrbitDB paused")
        })()
    }
    else if (message === "resumeOrbitDB") {
        (async () => {
            logger.info("Resuming OrbitDB")
            if (!ctx.ipfs) throw new Error("IPFS is not initialized")
            ctx.orbitdb = await getOrbitDB(true)
            await ctx.identity?.provider.keystore.open()
            ctx.registry = new UserRegistry(ctx.ipfs.api, ctx.orbitdb)
            await ctx.registry.open()
            logger.info("Replicating registry")
            await ctx.registry.replicate()
            ctx.shareLog = new IPFShareLog(ctx.ipfs.api, ctx.orbitdb, "ipfs-sharelog")
            await ctx.shareLog.open()
            await ctx.shareLog.onNewShare()
        })()
    }
  })
  socket.on("error", err => {
      logger.error("Error in OrbitDB service", err)
      throw err
  })
})

return server
}
```

4.3.4. Proceso demonio

```

public async startDaemon(): Promise<void> {
  const service = IPFSNodeManager.createOrbitDBService()
  service.listen(3000, "localhost", () => {
      logger.info("OrbitDB service listening on port 3000")
  })
  let daemon = await this.createNode()
  daemon = await daemon.start()
      .catch((err) => {
          logger.error("Error launching daemon", err)
          process.exit(1)
      })
  daemon.subprocess?.stdout?.pipe(process.stdout)
  daemon.subprocess?.stderr?.pipe(process.stderr)
  daemon.subprocess?.setMaxListeners(Infinity)
  logger.debug(`IPFS Daemon started with pid ${daemon.subprocess?.pid}`)
  ctx.ipfs = daemon
  ctx.orbitdb = await getOrbitDB(true)
  ctx.registry = new UserRegistry(ctx.ipfs.api, ctx.orbitdb)
  logger.info("Replicating registry")
  await ctx.registry.open()
  await ctx.registry.replicate()
  ctx.shareLog = new IPFShareLog(ctx.ipfs.api, ctx.orbitdb, "ipfs-sharelog")
  await ctx.shareLog.open()
  await ctx.shareLog.onNewShare()
  ctx.appConfig = await getAppConfigAndPromptIfUsernameInvalid(true)
  process.on("SIGINT", () => {
      (async () => {
          logger.info("Killing daemon")
          await daemon.stop()
          fs.rmSync(path.join(IPFSNodeManager.getRepoPath(), "api"), {force:true})
          await ctx.orbitdb?.disconnect()
          process.exit(0)
      })()
  })

  process.on("beforeExit", () => {
      (async () => {
          logger.info("Daemon exiting")
          await daemon.stop()
          fs.rmSync(path.join(IPFSNodeManager.getRepoPath(), "api"), {force:true})
          process.exit(0)
      })()
  })
```

```
}
public static async stopDaemon(): Promise<void> {
  // get the pid of the daemon
  const pid = await psList().then((list) => {
      const daemon = list.find((someProcess) =>
      ↪  someProcess.name.includes("node_modules/go-ipfs/bin/ipfs"))
      if (daemon === undefined) {
          return undefined
      }
      return daemon.pid
  })
  if (pid === undefined) {
      console.error("Could not find daemon")
      return
  }
  // kill the daemon
  process.kill(pid)
}
```

La invocaicón de este proceso se realiza mediante el comando `ipfshare daemon <action>`. Las acciones disponibles son:

- `start`: Inicia el proceso demonio.
- `stop`: Detiene el proceso demonio.

4.3.5. Registry

El Registry se ha representado en una clase abstracta que define la interfaz que debe implementar cualquier Registry que se quiera usar en el sistema. Esta clase abstracta se ha definido de la siguiente forma:

```
import { IPFS } from "kubo-rpc-client/dist/src/types"
import OrbitDB from "orbit-db"
import AccessController from "orbit-db-access-controllers/interface"
import DocumentStore from "orbit-db-docstore"
import { IdentityProvider } from "orbit-db-identity-provider"
// S es el tipo de Store de Orbit que se desee usar (DocumentStore, KeyValueStore, etc.)
// DocType es el tipo de documento que se va a almacenar en el Store
// AccessController es un controlador de acceso que debe implementar las políticas de acceso
↪  al Registry ya explicadas
export abstract class Registry<S, DocType> {
    abstract accessController: AccessController
    abstract store: S
    abstract open(): Promise<void>
    abstract create(): Promise<void>
    abstract replicate(): Promise<void>
    abstract close(): Promise<void>
    abstract addUser(user: DocType): Promise<void>
    abstract getUser(entryId: string): Promise<DocType | undefined>
    abstract updateUser(entryId: string, updates: Partial<DocType>): Promise<void>
    abstract searchUsers(queryFn: (entry: DocType) => boolean): Promise<DocType[]>
    abstract deleteUser(entryId: string): Promise<void>
}
```

Se ha implemntado un controlador de acceso básico '`IPFShareRegistryAccessController`' que implementa las políticas de acceso al Registry:

```
export class IPFShareRegistryAccessController extends AccessController{
  _orbitdb: OrbitDB
  _registry: Registry<any, any>

  constructor(orbitdb: OrbitDB, registry: Registry<any, any>) {
    super()
    this._orbitdb = orbitdb
    this._registry = registry
  }
```

```
  // este es el método al que llama OrbitDB para comprobar si se puede añadir una entrada al Registry
  async canAppend(entry: LogEntry<any>, identityProvider: IdentityProvider): Promise<boolean> {
    const userId = await entry.identity.id
    if (!userId) {
        throw new Error("identity is not set")
    }
    logger.debug(`Registry: ${userId} is trying to append`)
    const existingUser = await this._registry.getUser(userId)

    // Only allow appending to the log if the user does not exist or the user is the owner
    if (!existingUser || existingUser.orbitdbIdentity === entry.identity.id) {
        logger.debug(`Registry: ${userId} authorized to append`)
        // Allow access if identity verifies
        return true
    }
    logger.debug(`Registry: ${userId} NOT authorized to append`)
    return false
  }

  static get type(): string {
      return "ipfshare-registry"
  }

  save(): Promise<any> {
    return Promise.resolve({})
  }

  get address(): string {
    return this._registry.store.address.toString()
  }

  static async create(orbitdb: OrbitDB, options:{registry: Registry<any, any>}): Promise<AccessController> {
    // options must contain the registry to be used for access control
    return new IPFShareRegistryAccessController(orbitdb, options.registry)
  }
}
```

IPFShare implementa un Registry propio que se ha denominado '`UserRegistry`'. Las entradas en este Registry son de tipo `RegistryEntry`:

```
export interface RegistryEntry {
  peerId: string
  orbitdbIdentity: string // DID
  username: string // alias
}
```

```
export class UserRegistry implements Registry<DocumentStore<RegistryEntry>, RegistryEntry> {
store: DocumentStore<RegistryEntry>
orbitdb: OrbitDB
ipfs: IPFS
accessController: IPFShareRegistryAccessController

constructor(ipfs: IPFS, orbitdb: OrbitDB) {
  this.ipfs = ipfs
  this.orbitdb = orbitdb
  this.store = {} as DocumentStore<RegistryEntry>
  this.accessController = new IPFShareRegistryAccessController(orbitdb, this)
}

async open(): Promise<void> {
  try {
    this.store = await this.orbitdb.docstore<RegistryEntry>(
        "ipfshare-registry",
        {
            accessController: {
                type: IPFShareRegistryAccessController.type,
                registry: this,
            },
            // esta es la clave por la que se indexan los documentos del Registry
            indexBy: "orbitdbIdentity",
        }
```

```
26      )
27      await this.store.load()
28    } catch (e) {
29      // si falla open es porque la base de datos no existe, hay que crearla
30      await this.create()
31    }
32  }
33
34  async close(): Promise<void>{
35      await this.store.close()
36  }
37
38  async create(): Promise<void> {
39    try {
40      this.store =
41        await this.orbitdb.docstore("ipfshare-registry",
42          {
43            accessController: {
44                  type: IPFShareRegistryAccessController.type,
45                  registry: this,
46              },
47              create: true, // se añade esta opción para que se cree la base de datos si no
                ↪  existe
48              indexBy: "orbitdbIdentity",
49
50          }
51        )
52    }catch (e) {
53      logger.error(e)
54    }
55  }
56
57  // este método es llamado por el demonio para replicar el Registry
58  async replicate(): Promise<void> {
59    await this.store.load()
60    this.store.events.on("replicated", (address) => {
61        logger.info(`Registry replicated ${address} `)
62    })
63    this.store.events.on("replicate", (address) => {
64        logger.info(`Registry replicate ${address} `)
65    })
66    // cuando se conecta un nuevo nodo al sistema recargamos el Registry
67    this.store.events.on("peer", (peer) => {
68        (async () => {
69            await this.store.load()
70        })()
71        logger.info(`Registry peer connected ${peer}`)
72    })
73    this.store.events.on("replicate.progress", (address, hash, entry, progress, have) => {
74        logger.info(`Registry replication progress ${address}, ${hash}, ${entry}, ${progress},
            ↪  ${have}`)
75    })
76    this.store.events.on("peer.exchanged", (peer, address, heads) => {
77        logger.info(`Registry\n\tpeer ${peer} exchanged, ${heads.toString()}`)
78    } )
79  }
80
81  async addUser(user: RegistryEntry): Promise<void> {
82    if (!user.username) throw new Error("Username is not set")
83    await this.store.put(user)
84  }
85
86  async getUser(entryId: string): Promise<RegistryEntry | undefined> {
87    const matchingUsers = this.store.get(entryId)
88    if (matchingUsers.length === 0) return undefined
89    if (matchingUsers.length > 1) throw new Error("Multiple users with the same username")
90    return matchingUsers[0]
91  }
92
93  async updateUser(entryId: string, updates: Partial<RegistryEntry>): Promise<void> {
94    const oldEntry = await this.getUser(entryId)
95    if (!oldEntry) throw new Error("User not found")
96    const newEntry = {
97        ...oldEntry,
98        ...updates
99    } as RegistryEntry
100   this.store.put(newEntry)
101 }
```

```
102
103 async deleteUser(entryId: string): Promise<void> {
104     const user = await this.getUser(entryId)
105     if (!user) {
106         throw new Error("User not found")
107     }
108     await this.store.del(entryId)
109 }
110
111 async searchUsers(mapper: (entry: RegistryEntry) => boolean): Promise<RegistryEntry[]> {
112     const allUsers = this.store.query(mapper)
113     return allUsers
114 }}
```

4.3.6. Compartición de un archivo

En la CLI se ha implementado un comando para compartir archivos y carpetas. Este comando se ha implementado de la siguiente forma:

```
1  program.command("share")
2    .argument("[path...]", "Path to file or folder to upload")
3    .action(async (paths) => {
4      if (!paths || paths.length === 0) {
5          program.help()
6      }
7      await withContext(async () => {
8        if (!ctx.did) throw new Error("DID not initialized")
9        const { encryptedStream, iv, key } = await createEncryptedTarFromPaths(paths)
10       const res = await uploadToIpfs(encryptedStream)
11       console.log(`Uploaded to IPFS with CID: ${res.cid.toString()}, size: ${res.size}`)
12       const recipients: string[] = await interactiveRegistryPrompt("Select recipients")
13       recipients.push(ctx.appConfig!.user.orbitdbIdentity)
14       console.log(`Recipient DIDs ${recipients}`)
15       const share: Share = {
16           contentCID: res.cid,
17           iv: iv.toString("base64"),
18           key: key.toString("base64"),
19           recipientDIDs: recipients
20       }
21       const shareCID = await addEncryptedObject(share) // añadir el objeto cifrado a OrbitDB
22       const msg = await messagePrompt() // mensaje que el usuario contenido en el Share
         ↪  (puede ser cualquier cosa)
23       const recipientNames= getRecipientNames(share.recipientDIDs) // nombres de los
         ↪  destinatarios
24       // se añade la compartición al ShareLog, como se puede observar se pueden añadir las
         ↪  entradas que se deseen al objeto Share según las necesidades de la implementación
         ↪  del ShareLog
25       const shareHash = await ctx.shareLog?.addShare(
26           {
27               message: msg,
28               recipients: recipients,
29               shareCID: shareCID,
30               senderName: ctx.appConfig!.user.username,
31               recipientNames: recipientNames,
32               senderId: ctx.appConfig!.user.orbitdbIdentity
33           }
34       )
35       console.log(`Added share to ShareLog with hash: ${shareHash}`)
36       console.log(`Content CID: ${res.cid.toString()}`)
37       console.log(`Share CID: ${shareCID.toString()}`)
38     })
39 })
```

''*interactiveRegistryPrompt*' es un método que muestra un diálogo interactivo para seleccionar los usuarios con los que se quiere compartir el archivo. Este diálogo se ha implementado con la biblioteca `prompts`.

```
1 const interactiveRegistryPrompt:  (promptMessage: string)  => Promise<string[]> = async
  ↪  (promptMessage) => {
2   const users =  (await ctx.registry?.searchUsers(() => true))
```

```
  if (!users) throw new Error("No users found")
  const choices = users.filter(user => ctx.appConfig?.user.peerId !== user.peerId)
      .map((user) => {
          const res: prompts.Choice = {
              title: `${user.username} - ${user.orbitdbIdentity} - peerId: ${user.peerId}`,
              value: user.orbitdbIdentity,
          }
          return res
      })
  // este prompt implementa un diálogo interactivo para seleccionar los usuarios con los que se quiere compartir el archivo
  const response = await prompts.prompt({
      type: "autocompleteMultiselect",
      name: "recipients",
      message: promptMessage,
      choices: choices,
      min: 0,
      max: users.length,
      hint: "- Space to de/select. Return to submit"
  })

  if (response.recipients) {
      return response.recipients
  } else {
      throw new Error("No recipients selected")
  }
}

function getRecipientNames(recipients: string[]): string[] {
  if (!ctx.registry) throw new Error("Registry not initialized")
  const users = ctx.registry.store.query((reg) => {
      return recipients.includes(reg.orbitdbIdentity)
  })
  const recipientNames = users.map((user) => user.username)
  return recipientNames
}
```

Esto es solo un ejemplo de las posibilidades que ofrece el Registry. Los datos y metadatos, además de la complejidad dependen de la la implementación de este. El concepto de Registry es muy flexible y se puede adaptar a cualquier necesidad (dentro de unas limitaciones).

Los métodos '*createEncryptedTarFromPaths*', '*uploadToIpfs*', '*addEncryptedObject*' implementan la secuencia de compartición descrita en 4.2.5: 'Protocolo de compartición'.

```
// este método crea un archivo tar comprimido y cifrado con AES-256-CBC a partir de una lista de archivos y carpetas
// este targz encriptado se devuelve como un stream
export async function createEncryptedTarFromPaths(pathsToInclude: string[], tarPath="test.tar.gz") {
  const progressBarArchive = new SingleBar(
      {
          format: "Archive creation [{bar}] {percentage}% | ETA: {eta}s | {value}/{total}",
      },
      Presets.shades_classic
  )
  const progressBarEncryption = new SingleBar(
      {
          format: "Encryption [{bar}] {percentage}% | ETA: {eta}s | {value}/{total}",
      },
      Presets.shades_classic
  )
  // Generate a random key
  const key = crypto.randomBytes(32) // This generates a 256-bit key
  // console.log("Encryption key:", key.toString("hex"))
  const iv = crypto.randomBytes(16)
  // console.log("Encryption iv:", iv.toString("hex"))
  // Create a cipher using the random key
  const cipher = crypto.createCipheriv("aes-256-cbc", key, iv)
  // Create a pass-through stream which will be given to ifps.add
  const pass = new PassThrough()

```

```
  // tar archive stream with gzip compression
  const archive = archiver("tar",
      {
          gzip: true,
          zlib: { chunkSize: 10 * 1024 * 1024, level: 1}, // Chunk size of 10MB
          gzipOptions: { chunkSize: 10 * 1024 * 1024, level: 1 }, // Chunk size of 10MB,
          statConcurrency: 50,
      }
  )
  // good practice to catch warnings (ie stat failures and other non-blocking errors)
  archive.on("warning", function(err) {
      if (err.code === "ENOENT") {
          console.warn("File not found:", err)
          process.exit(1)
      }
      throw err
  })
  archive.on("entry", function (entry) {
      progressBarArchive.increment()
      progressBarArchive.updateETA()
  })
  archive.on("progress", function (progress) {
      console.log(`Progress ${progress.entries.processed} / ${progress.entries.total}`)
  })
  // good practice to catch this error explicitly
  archive.on("error", function(err) {
      throw err
  })
  archive.on("end", function () {
      console.log("Tar file created")
      progressBarArchive.stop()
      console.log("Archive finished: ")
      console.timeEnd("Archive")
      console.log("\n")
      console.time("Encryption")
      progressBarEncryption.start(progressBarArchive.getTotal(), 0)
  })
  let entries = 0
  console.time("Archive")
  const statPromises = pathsToInclude.map(async (p) => {
      const stat = await fsp.stat(p)
      if (stat.isDirectory()) {
          const files = await fsp.readdir(p)
          entries += files.length
          archive.directory(p, path.basename(p))
      } else if (stat.isFile()) {
          entries += 1
          archive.file(p, { name: path.basename(p) })
      }
  })
  progressBarArchive.start(entries, 0)
  await Promise.all(statPromises)
  // Finalize the archive (ie. we are done appending files but streams have to finish yet)
  cipher.on("data", (chunk) => {
      progressBarEncryption.increment(chunk.length)
  })
  cipher.on("end", function () {
      progressBarEncryption.stop()
      console.log("Encryption finished")
      console.timeEnd("Encryption")
      console.log("\n")
  })
  // Pipe the archive to the cipher and then to the pass-through stream
  archive.pipe(cipher).pipe(pass)
  // Finalize the archive (ie. we are done appending files but streams have to finish yet)
  archive.finalize()
  return { encryptedStream: pass, iv: iv, key: key }
}

// este método sube un stream a IPFS y devuelve el CID
export async function uploadToIpfs(stream: PassThrough): Promise<AddResult> {
  if (!ctx.ipfs) throw new Error("IPFS not initialized")
  // se debe añadir la opción pin:true para evitar el recolector de basura de IPFS borre el
  ↪  archivo
  const addRes = await ctx.ipfs.api.add(stream, {pin: true})
  return addRes
}
```

```
// formato de un share, este es el objeto que se cifra en un JWE, se sube a IPFS y se comparte en el ShareLog
export type Share = {
  contentCID: CID,
  key: string,
  iv: string,
  recipientDIDs: string[],
}

// este método cifra un objeto de tipo Share en un JWE y lo sube al DAG de IPFS
export async function addEncryptedObject (share: Share): Promise<CID> {
  if (!ctx.did) throw new Error("DID not initialized")
  if (!ctx.ipfs) throw new Error("IPFS not initialized")
  if (!ctx.ipfs.api) throw new Error("IPFS API not initialized")
  const jwe = await ctx.did.createDagJWE(share, share.recipientDIDs)
  const result = await ctx.ipfs.api.dag.put(jwe,
    {
      storeCodec: "dag-jose", // como se había mencionado debemos usar el codec dag-jose para guardar JWE en el DAG
      hashAlg: "sha2-256",  // el algoritmo de hash que se usa para calcular el CID
      pin: true  // de nuevo, se debe añadir la opción pin:true para evitar el recolector de basura de IPFS borre el archivo
    })
    .catch((err) => {
        console.error(err)
        throw err
    })
  if (!result) throw new Error("IPFS add failed")
  return result
}
```

4.3.7. Acceso a un archivo compartido

Cuando un usuario recibe una notificación de una compartición por parte de otro usuario con este, puede descargar el archivo compartido. Para ello se ha implementado el siguiente comando:

```
program.command("download")
  .summary("Print the contents of a given CID")
  .description("Prints the contents of a given CID. The CID must be an encrypted jwt.")
  .argument("[cids...]", "CID of file or folder to cat")
  .addOption(new Option("-o, --output <path>", "Output path"))
  .action(async (cids: string[], command) => {
    if (!cids || cids.length === 0) {
      console.log("No cids provided")
      program.help()
      process.exit(1)
    }
    if (command.output == null || command.output === undefined || command.output === "") {
      console.log("No output path provided")
      program.help()
      process.exit(1)
    }
    const outPath = command.output
    fs.mkdirSync(outPath, { recursive: true })
    await withContext(async () => {
      for (const strCid of cids) {
        const cid = CID.parse(strCid)
        const share = await getEnctrypedObject(cid)
        // accede al JWE cifrado en el DAG de IPFS
        console.log(share)
        // extrae el contenido del JWE
        const {contentCID, iv, key} = share
        console.log(`Content CID: ${contentCID.toString()}}`)
        console.log(`iv: ${Buffer.from(iv, "base64").toString("hex")}`)
        console.log(`key: ${Buffer.from(key, "base64").toString("hex")}`)
        // descarga el contenido cifrado de IPFS
        const contentRes = await downloadFromIpfs(contentCID)
        const contentStats = await ctx.ipfs!.api.files.stat("/ipfs/" + contentCID.toString())
        // desencripta el contenido usando la clave y el vector de inicialización (iv) usados para el cifrado.
```

```
34          await decryptAndExtractTarball(contentRes, Buffer.from(key, "base64"),
            ↪  Buffer.from(iv, "base64"), outPath,contentStats.size)
35        }
36      })
37    })
38
39  // descarga el archivo cifrado y retorna un stream para su consumo
40  export async function downloadFromIpfs(cid: CID): Promise<NodeJS.ReadableStream> {
41    if (!ctx.ipfs) throw new Error("IPFS not initialized")
42    const asyncIterable = ctx.ipfs.api.cat(cid)
43    const stream = Readable.from(asyncIterable)
44    return stream
45  }
46
47  // recibe un stream del archivo encriptado, lo desencripta y extrae el tar en el directorio
    ↪  de salida
48  export function decryptAndExtractTarball(encryptedStream: NodeJS.ReadableStream, key:
    ↪  Buffer, iv: Buffer, outputPath: string, totalSize: number){
49    return new Promise<void>((resolve, reject) => {
50      // Create a decipher using the provided key and iv
51      const decipher = createDecipheriv("aes-256-cbc", key, iv)
52      const untar = tar.extract({ cwd: outputPath })
53      // Create a progress bar for the decryption and extraction process
54      const progressBar = new SingleBar({format: "progress [{bar}] {percentage}% | ETA: {eta}s
        ↪  | {value}/{total}"}, Presets.shades_classic)
55      const totalSizeMb = Math.round(totalSize / 1_048_57)
56      progressBar.start(totalSizeMb, 0)
57      untar.on("data", (chunk) => {
58          progressBar.increment(Math.round(chunk.length / 1_048_57))
59          progressBar.updateETA()
60      })
61      untar.on("finish", () => {
62          console.log("Extraction finished")
63          progressBar.stop()
64          resolve()
65      })
66      untar.on("error", (error) => {
67          progressBar.stop()
68          reject(error)
69      })
70      encryptedStream
71        .pipe(decipher)
72        .pipe(untar)
73    })
74  }
```

4.4. Descarga, instalación y modo de uso de IPFShare

4.4.1. Descarga e instalación

Para descargar el programa se puede clonar el repositorio de GitHub, construir el proyecto y ejecutarlo. Para ello se necesita tener instalado *Node.js* y *yarn*. Es necesario usar yarn debido a que es el único manejador de paquetes de NodeJS capaz de resolver las dependencias en todas las plataformas.

```
1  $ git clone https://github.com/nicocossiom/IPFShare && cd IPFShare && yarn install && yarn
   ↪  build
```

Si se elige está opción todos los comandos deben ejecutarse precedidos de *yarn start*. Por ejemplo: *yarn start download <CID>*

La forma recomendad de instalación es añadiendo de forma global el paquete de IPFShare en sí mismo, esto permite acceder a IPFShare con el comando *ipfshare*, sin necesidad de ser precedido de *yarn start*. Para instalar de forma global se debe ejecutar el siguiente comando:

```
$ yarn global add git+https://github.com/nicocossiom/IPFShare.git
```

Después de instalar IPFShare se debe ejecutar el comando *setup* para configurar el programa. Este comando crea el directorio donde se guardarán todos los datos usados por el programa. Este directorio se puede cambiar mediante la variable de entorno *IPFSHARE_HOME*. Por defecto se usa el directorio *$HOME/.ipfshare*.

Ejemplo:

```
$ ipfshare setup
Where would you like to store your IPFShare config? (enter for /Users/pepperonico/.ipfshare)
Setting up IPFShare in /Users/pepperonico/.ipfshare
Created folder and IPFShare home set to "/Users/pepperonico/.ipfshare"
IPFShare setup done
```

4.4.2. Modo de uso

IPFShare requiere tener en ejecución el demonio para el correcto funcionamiento del sistema. Esto se debe a que el demonio debe replicar y recibir entradas de las basses de datos de OrbitDB, además de mantener accesibles las comparticiones hechas por el usuario para otros usuarios del sistema.

La ejecución del demonio puede hacerse en segundo plano o en primer plano.
Para ejecutarlo en primer plano se debe ejecutar el comando *ipfshare daemon start*.
Para ejecutarlo en segundo plano se debe ejecutar el comando *ipfshare daemon start &*.
Para detener el demonio se debe ejecutar el comando *ipfshare daemon stop* o matar el proceso del mismo con kill o pkill ipfshare.

Ejemplo de uso:

```
$ ipfshare daemon start
2023-07-09 14:34:36 info: New repo crated
2023-07-09 14:34:36 debug: Spawning node, current number of nodes: 0
2023-07-09 14:34:36 debug: Selected repo /Users/pepperonico/.ipfshare/ipfsRepo
2023-07-09 14:34:36 info: OrbitDB service listening on port 3000
2023-07-09 14:34:37 debug: IPFS Daemon started with pid 73123
2023-07-09 14:34:37 debug: orbitdbPath: /Users/pepperonico/.ipfshare/orbitdb
2023-07-09 14:34:37 debug: Custom registry access added: true
2023-07-09 14:34:37 debug: DID authenticated:
↪  did:key:z6Mksf6J4uriMc8PADNUzoFi612Z2acfckqLjjo5bYKhufQ6
2023-07-09 14:34:37 info: OrbitDB instance created:
↪  did:key:z6Mksf6J4uriMc8PADNUzoFi612Z2acfckqLjjo5bYKhufQ6
2023-07-09 14:34:37 info: Replicating registry
2023-07-09 14:34:37 debug: IPFShareLog created with address ipfs-sharelog
2023-07-09 14:34:37 info: ShareLog loaded
↪  /orbitdb/zdpuAvtf7yawnNwu4Ygxg32DzdGmCf4r6WrcA4jwrJLFtjA81/ipfs-sharelog
  Username is not set or not registered
```

Como se puede observar, el demonio avisa de que este usuario aún no se ha registrado. Para registrarse en el Registry se puede ejecutar cualquier comando de la CLI. Por ejemplo:

```
$ ipfshare registry list
  2023-07-09 14:37:00 debug: Spawning node, current number of nodes: 0
  2023-07-09 14:37:00 debug: Selected repo /Users/pepperonico/.ipfshare/ipfsRepo
  2023-07-09 14:37:00 info: Connected to the daemon process.
  2023-07-09 14:37:00 debug: Sent message pauseOrbitDB
  2023-07-09 14:37:00 debug: Received response for message pauseOrbitDB:
```

```
2023-07-09 14:37:00 debug: orbitdbPath: /Users/pepperonico/.ipfshare/orbitdb
2023-07-09 14:37:00 debug: Custom registry access added: true
2023-07-09 14:37:00 debug: DID authenticated:
↪  did:key:z6Mksf6J4uriMc8PADNUzoFi612Z2acfckqLjjo5bYKhufQ6
2023-07-09 14:37:00 info: OrbitDB instance created:
↪  did:key:z6Mksf6J4uriMc8PADNUzoFi612Z2acfckqLjjo5bYKhufQ6
2023-07-09 14:37:00 debug: IPFShareLog created with address ipfs-sharelog
Username is not set or not registered
? Enter a username Nico Cossio
Username Nico Cossio is not in the registry
Setting username to Nico Cossio
2023-07-09 14:37:08 debug: Registry:
↪  did:key:z6Mksf6J4uriMc8PADNUzoFi612Z2acfckqLjjo5bYKhufQ6 is trying to append
2023-07-09 14:37:08 debug: Registry:
↪  did:key:z6Mksf6J4uriMc8PADNUzoFi612Z2acfckqLjjo5bYKhufQ6 authorized to append
Current user: {
  "username": "Nico Cossio",
  "orbitdbIdentity": "did:key:z6Mksf6J4uriMc8PADNUzoFi612Z2acfckqLjjo5bYKhufQ6",
  "peerId": "12D3KooWRowWKkzKfDc1m1C4Be4Rq7RGYAJGiUKMmnNsQ7sHfa26"
}
Registry:
[
  {
    username: 'Nico Cossio',
    orbitdbIdentity: 'did:key:z6Mksf6J4uriMc8PADNUzoFi612Z2acfckqLjjo5bYKhufQ6',
    peerId: '12D3KooWRowWKkzKfDc1m1C4Be4Rq7RGYAJGiUKMmnNsQ7sHfa26'
  }
]
```

El usuario se ha registrado correctamente con el nombre de usuario elegido. Esta información se almacena en el archivo *config.json* en el directorio de IPFShare. Cuando se inicia cualquier comando de la CLI, este lee el archivo *config.json* y obtiene del Registry la entrada correspondiente al DID del usuario. Si el usuario no está registrado en el Registry, se le pedirá que se registre. Si está se verifica que esta entrada pertenece al usuario que está ejecutando el comando.
Aunque a efectos prácticos una entidad maligna podría intentar impersonar a otro usuario del sistema, esto no es posible por las razones explicadas en la subseccción 4.2.3: 'Autenticación y control de acceso'.

Una vez el usuario ha realizado estos pasos puede ejectuar cualquier comando de la CLI. Para más información sobre los comandos disponibles y su método de uso puede consultar el comando de ayuda: *ipfshare help*.

```
$ ipfshare --help
```

```
,,
`7MMF'`7MM"""Mq.`7MM"""YMM  .M"""bgd `7MM
MM    MM   `MM. MM    `7 ,MI    "Y   MM
MM    MM   ,M9  MM   d   `MMb.       MMpMMMb.   ,6"Yb.  `7Mb,od8 .gP"Ya
MM    MMmmdM9   MM""MM     `YMMNq.   MM    MM  8)   MM    MM' "',M'   Yb
MM    MM        MM   Y   .     `MM   MM    MM   ,pm9MM    MM    8M""""""
MM    MM        MM       Mb     dM   MM    MM  8M   MM    MM    YM.    ,
.JMML..JMML.    .JMML.     P"Ybmmd"  .JMML  JMML.`Moo9^Yo..JMML.   `Mbmmd'

An IPFS-based, encrypted file sharing CLI tool
  Usage: ipfshare [options] [command]

  Options:
  -V, --version                output the version number
  -h, --help                   display help for command
```

```
  Commands:
  setup [path]                  Run initial setup
  daemon <action>               Start the Kubo (go-ipfs) daemon.
   ↪  This is a custom daemon for IPFShare. See daemon --help for
   ↪  more
  info.
  cat [cids...]                 Print the contents of a given CID
  ls [cids...]                  List the contents of a given CID
  share [path...]
  download [options] [cids...]  Print the contents of a given CID
  sharelog                      Interact with the global share log
  shared                        List all files and folders shared
   ↪  with you
  shares                        Interact with shares created by you
  registry                      Access the IPFShare global registry.
   ↪  Change username or delete account.
  info                          Provides information about the
   ↪  running ipfshare instance such as DID and peerID
```

Para obtener más información sobre un comando especifico puede ejecutar el comando de ayuda de dicho comando. Por ejemplo:

```
$ipfshare registry  --help
  Usage: ipfshare registry [options] [command]

  Update and query the IPFShare global registry

  Options:
    -h, --help         display help for command

  Commands:
    update <username>  Update username if available
    delete             Delete your IPFShare account
    list               List all users in the IPFShare global
     ↪  registry
    register           Register yourself in IPFShare global registry
    search [options]   Search for a user in the IPFShare global
     ↪  registry
```

Capítulo 5

Evaluación de la implementación

En este capítulo se evaluará la implementación realizada en base a los requisitos definidos en el capítulo 4: 'Desarrollo de IPFShare'.

Para ello se ha creado un escenario de uso real con distintos usuarios en varios lugares del mundo. Este escenario se ha utilizado para realizar una serie de pruebas que permitan evaluar el sistema desarrollado. Para ello se han usado varios servidores proporcionados por Github Codespaces. Estos servidores corren instancias de IPFShare emulando un caso de uso real con usuarios en distintas ubicaciones geográficas.

Las pruebas realizadas son:

- **Registro e interconexión:** se comprueba que los usuarios pueden registrarse en el sistema, a medida que cada usuario se registra el Registry debe actualizar su lista de usuarios, esto debe verse reflejado en todos los clientes.
- **Compartición de un archivo con un usuario específico**: un usuario comparte un archivo con un usuario específico, este debe recibir una notificación y poder descargar el archivo.
- **Compartición de un archivo con todos los usuarios registrados**: un usuario comparte un archivo con todos los usuarios registrados, estos deben recibir una notificación y poder descargar el archivo.
- **Compartición de un archivo por parte de un usuario, intento de descarga por parte de un usuario no autorizado**: debe fallar la descarga al estar este usuario no autorizado.
- **Intento de manipulación del Registry - modificación y eliminación de una entrada**: se intenta modificar y eliminar una entrada del Registry por parte de una entidad no autorizada, debe fallar.
- **Intento de un usuario de tener varias entradas en el Registry:** se intenta registrar un usuario con un DID que ya está registrado, debe fallar.
- **Comparticiones realizadas por el usuario**: El usuario debe poder acceder a la lista de archivos que ha compartido.
- **Comparticiones recibidas por el usuario**: El usuario debe poder acceder a la lista de archivos que ha recibido.

No se han incluido pruebas sobre funcionalidades de grupos debido a que no se han

implementado por falta de tiempo. Aunque cabe destacar que bajo la infraestructura actual podrían ser fácilmente implementadas en un futuro.

Estas pruebas están documentadas en un vídeo que se puede encontrar en el siguiente enlace: https://youtu.be/2Z3Q4Z3Z8ZM.

Capítulo 6

Resultados y conclusiones

Se han cumplido casi todos los objetivos del proyecto enumerados en 1.2: Objetivos y alcance del proyecto. Se ha desarrollado un sistema de intercambio de ficheros basado en IPFS, mediante una aplicación de escritorio. Este sistema permite a los usuarios compartir archivos de forma segura y confiable, sin necesidad de ningún proveedor central de ningún tipo. Integra capacidades de encriptación y control de acceso para garantizar la seguridad de los archivos compartidos. La integración de cuentas de usuario, aunque sin la posibilidad de hacer grupos. También se puede elegir contactos con los que compartir. Por último se integra un sistema de notificaciones para el que los usuarios puedan recibir avisos de nuevos archivos compartidos, o de cambios en los archivos compartidos.

El objetivo de investigar sobre IPFS y su funcionamiento para entender cómo funciona el protocolo y cómo se puede utilizar para el sistema propuesto también se ha cumplido. Se ha realizado un estudio con una profundidad adecuada al ámbito y alcance de este proyecto. Se han explicado conceptos de IPFS, su funcionamiento, arquitectura, algoritmo de intercambio de bloques, identificación basada en contenido, hasta su estructura de datos. Con estos conceptos como base se ha profundizado en el ecosistema en torno a IPFS, con objetivo de comprender la madurez y viabilidad de esta tecnología, así como de las herramientas basadas en esta que se pueden utilizar para el sistema propuesto. Sobre esto último:

Tras el satisfactorio resultado obtenido se podría pensar que IPFS y su ecosistema están lo suficientemente avanzados como para ser utilizados en un sistema de producción. Sin embargo, tras el desarrollo de este proyecto se ha podido comprobar que IPFS y su ecosistema aún están en una fase temprana de desarrollo. Esto se debe a que se han encontrado varios problemas durante el desarrollo del proyecto, algunos de ellos se han podido solucionar, pero otros no. Estos problemas se han debido principalmente a la falta de documentación y a la inmadurez o falta de mantenimiento de algunas herramientas.

Uno de los grandes obstáculos en el entorno de NodeJS para IPFS es la falta de soporte y mantenimiento de los paquetes de tipos de Typescript. Ciertos paquetes como *orbitdb-identity-provider* tienen sus paquetes de tipos desactualizados, están erróneamente implementados u ocurre lo mismo con la documentación que proveen. Para solventar estos problemas se ha tenido que recurrir al uso de *//@ts-ignore* en

múltiples ocasiones, esta es una directiva que indica al compilador de Typescript que haga caso omiso a los errores de tipos que se producen en la línea siguiente. Este tipo de directivas está generalmente desaconsejado y hace que el uso de Typescript pierda sentido al desarrollar, ya que no realizan las comprobaciones de tipado.

Para el paquete de *orbitdb-identity-provider* se ha tenido que recurrir a hacer *fork* del paquete para poder arreglar dos errores para el proveedor de identificación basado en DIDs, que es el usado por IPFShare.

Esta falta o mala documentación también se traslada a las propias implementaciones de IPFS. Un ejemplo de esto es la discusión que se dio en el foro de IPFS sobre la posibilidad de compartir el MFS (Mutable File System) de un nodo a otro [25]. Varios usuarios expresaron su frustración y confusión sobre cómo usar el MFS, cómo acceder a los archivos desde otros nodos, cómo sincronizar los cambios y cómo resolver los conflictos. La documentación oficial de IPFS no ofrece mucha claridad sobre estos aspectos, y tampoco hay muchos ejemplos o tutoriales que expliquen cómo usar el MFS de forma efectiva. Esta clase de situaciones hace hace que sea difícil para los desarrolladores aprovechar el potencial de IPFS como un sistema de archivos distribuido y mutable. Este proyecto podría haberse implementado en Go, pero JavaScript y el entorno de NodeJS son tremendamente más populares entre desarrolladores.

Capítulo 7

Trabajos futuros

Bibliografía

[1] "IPFS Powers the Distributed Web." [Online]. Available: https://ipfs.tech/

[2] "Web3," *Wikipedia*, Apr. 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Web3&oldid=1148197462

[3] P. Výboch, "Peer-to-peer protocols for file sharing: BitTorrent," Jun. 2017.

[4] "BitTorrent Protocol." [Online]. Available: http://www.bittorrent.org/beps/bep_0003.html

[5] J. Benet, "IPFS - Content Addressed, Versioned, P2P File System," Jul. 2014. [Online]. Available: http://arxiv.org/abs/1407.3561

[6] "Protocol Wars," *Wikipedia*, Mar. 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Protocol_Wars&oldid=1145543147

[7] B. Edwards, "The Foundation of the Internet: TCP/IP Turns 40," Sep. 2021. [Online]. Available: https://www.howtogeek.com/751880/the-foundation-of-the-internet-tcpip-turns-40/

[8] B. M. Leiner, V. G. Cerf, D. D. Clark, R. E. Kahn, L. Kleinrock, D. C. Lynch, J. Postel, L. G. Roberts, and S. Wolf, "A Brief History of the Internet," 1999. [Online]. Available: https://arxiv.org/abs/cs/9901011

[9] E. Mori, "Peter Kirstein obituary," *The Guardian*, Feb. 2020. [Online]. Available: https://www.theguardian.com/technology/2020/feb/09/peter-kirstein-obituary

[10] "Hiperenlace," *Wikipedia, la enciclopedia libre*, Jul. 2023. [Online]. Available: https://es.wikipedia.org/w/index.php?title=Hiperenlace&oldid=152311621

[11] "Protocol Labs." [Online]. Available: https://protocol.ai/

[12] "What is libp2p." [Online]. Available: https://docs.libp2p.io/concepts/introduction/overview/

[13] P. Labs, "Libp2p Connectivity." [Online]. Available: https://connectivity.libp2p.io/microgen.vercel.app

[14] "AutoNAT." [Online]. Available: https://docs.libp2p.io/concepts/nat/autonat/

[15] "Circuit Relay." [Online]. Available: https://docs.libp2p.io/concepts/nat/

circuit-relay/

[16] "Rendezvous." [Online]. Available: https://docs.libp2p.io/concepts/discovery-routing/rendezvous/

[17] "IPLD Docs." [Online]. Available: https://ipld.io/docs/

[18] "Peergos/Peergos: A p2p, secure file storage, social network and application protocol." [Online]. Available: https://github.com/Peergos/Peergos

[19] Filecoin, "A decentralized storage network for humanity's most important information." [Online]. Available: https://filecoin.io/

[20] "Tabcat/orbit-db-fsstore: A custom orbit-db store representing a file system." [Online]. Available: https://github.com/tabcat/orbit-db-fsstore

[21] "Sailplane-node," cypsela, May 2023. [Online]. Available: https://github.com/cypsela/sailplane-node

[22] A. Chakraborty, "System Design Analysis of Google Drive," Oct. 2020. [Online]. Available: https://towardsdatascience.com/system-design-analysis-of-google-drive-ca3408f22ed3

[23] SystemDesign, "System Design Interview: Dropbox or a Similar File Storage & Sharing Service (Google Drive/. . . ," Mar. 2023. [Online]. Available: https://medium.com/double-pointer/system-design-interview-dropbox-or-a-similar-file-storage-sharing-service-google-drive-3491

[24] "Kubo RPC API | IPFS Docs." [Online]. Available: https://docs.ipfs.tech/reference/kubo/rpc/#getting-started

[25] "Is it possible to share the MFS from one node to another - Help." [Online]. Available: https://discuss.ipfs.tech/t/is-it-possible-to-share-the-mfs-from-one-node-to-another/10513/8

[26] A. Das, "TCP/IP Protocol Architecture Model - How Does it Work?" Nov. 2022. [Online]. Available: https://geekflare.com/tcp-ip-protocol-architecture-model/

[27] "Protocolo de control de transmisión," *Wikipedia, la enciclopedia libre*, Feb. 2023. [Online]. Available: https://es.wikipedia.org/w/index.php?title=Protocolo_de_control_de_transmisi%C3%B3n&oldid=149440564

[28] "TCP/IP Model vs. OSI Model | Similarities and Differences." [Online]. Available: https://www.fortinet.com/resources/cyberglossary/tcp-ip-model-vs-osi-model

[29] M. Cooney, "SNA and OSI vs. TCP/IP," Oct. 2007. [Online]. Available: https://www.networkworld.com/article/2287941/sna-and-osi-vs--tcp-ip.html

[30] "Layers of OSI Model," Aug. 2017. [Online]. Available: https://www.geeksforgeeks.org/layers-of-osi-model/

Anexo

Característica	**IP/TCP**	**OSI**	**X.25**	**SNA**
Modelo	Suite de protocolos	Modelo de referencia	Protocolo de enlace	Suite de protocolos
Capas	4 (TCP/IP)	7	3	7
Año de lanzamiento	1974 (TCP) / 1981 (IP)	1984	1976	1974
Enfoque	Conmutación de paquetes	Conmutación de paquetes y circuitos	Conmutación de circuitos	Conmutación de paquetes y circuitos
Estándar	IETF	ISO	CCITT (ahora ITU-T)	IBM
Orientación	Red global	Interoperabilidad	Redes de área amplia (WAN)	Redes empresariales
Funcionalidades	Transmisión de datos, enrutamiento, control de flujo, control de congestión, conexión y desconexión	Transmisión de datos, enrutamiento, control de flujo, control de congestión, conexión y desconexión, servicios de presentación y aplicación	Transmisión de datos, control de flujo, conexión y desconexión	Transmisión de datos, enrutamiento, control de flujo, control de congestión, conexión y desconexión, servicios de presentación y aplicación
Uso en los años 90	Muy popular, base del Internet	Intento de reemplazar a TCP/IP, pero fracasó en la adopción generalizada	Utilizado en redes de área amplia (WAN), especialmente en Europa	Utilizado en redes empresariales, especialmente en sistemas mainframe de IBM

Descripción	Un modelo que se basa en la suite de protocolos TCP/IP para transmitir datos por Internet. El modelo es más simple y flexible que el modelo OSI y se usa ampliamente en la actualidad.	Un modelo que se basa en la suite de protocolos OSI para estandarizar la comunicación entre sistemas abiertos. El modelo segmenta múltiples funciones que el modelo IPTCP agrupa en capas únicas y define los servicios e interfaces para cada capa.	Un modelo que se basa en la suite de protocolos X.25 para proporcionar una conexión virtual entre terminales y computadoras a través de una red pública de conmutación de paquetes. El modelo fue uno de los primeros en ofrecer una comunicación confiable entre dispositivos remotos, pero ha sido reemplazado por tecnologías más rápidas y eficientes como Frame Relay e IP.	Un modelo que se basa en la suite de protocolos SNA para integrar los recursos informáticos distribuidos en una red jerárquica. El modelo fue desarrollado por IBM para conectar sus sistemas mainframe y periféricos, pero ha perdido popularidad frente a los modelos basados en IP.

Cuadro 1: Comparación de IP/TCP, OSI, X.25 y SNA en los años 90. Fuentes: [26], [27], [6], [28] [29] [30]