Informe de Trabajo Profesional

Análisis de ecosistemas para la implementación de plataformas como servicio para despliegue de aplicaciones comunitarias, distribuidas y descentralizadas

Integrantes

Lucas Nahuel Sotelo Guerreño
102730
lsotelo@fi.uba.ar

Sebastian Bento Inneo Veiga
100998
sinneo@fi.uba.ar

Joaquín Matías Velazquez
105980
jvelazquez@fi.uba.ar

Joaquín Prada
105978
jprada@fi.uba.ar

Tutor
Ariel Scarpinelli
ascarpinelli@fi.uba.ar

Índice

1. Resumen

En el siguiente trabajo se analizan distintos ecosistemas y tecnologías que se pueden utilizar para el despliegue de aplicaciones web comunitarias de manera distribuida y descentralizada.

Mediante tres casos de uso que ilustran diferentes características: un sitio web informativo, un repositorio de conocimiento y un mensajero en tiempo real, se comparan ventajas y desventajas del despliegue de cada uno de ellos en IPFS, blockchain y Hyphanet/Freenet, así como también se documenta el proceso del mismo.

2. Palabras Clave

Distribuido. Sistema. Comunitario. Descentralizado. Aplicación.

3. Abstract

The following work analyzes different ecosystems and technologies that can be used to deploy community web applications in a distributed and decentralized manner.

Using three use cases that illustrate different features: an informational website, a knowledge repository and a real-time messenger, the advantages and disadvantages of deploying each of them on IPFS, blockchain and Hyphanet/Freenet are compared, as well as the process to do so is documented.

4. Keywords

Distributed. System. Community. Decentralized. Application.

5. Introducción

Hoy en día, al querer desplegar una aplicación o sitio web comunitario, lo más común es hacerlo a través de un servicio de alojamiento (AWS, Azure, Google Cloud, entre otros) por la comodidad y facilidad que estas ofrecen, alquilando sus servidores para guardar y procesar datos.

Esto puede llegar a traer problemas para este tipo de aplicaciones. Uno de estos problemas puede ser monetario, ya que muchas veces estas aplicaciones dependen de donaciones o voluntarios para sustentarse, como es el caso de Wikipedia. Como también puede suceder que se encuentre en una zona de censura, lo cual facilita su bloqueo al ser servicios centralizados; entre otros problemas más.

Sin embargo, existen otros ecosistemas alternativos que se asemejan mucho más a la filosofía de estas aplicaciones, y que ayudan a combatir estos problemas. En donde las aplicaciones pueden estar alojadas por sus propios usuarios, donando su computo o espacio, y así logrando una descentralización.

En el siguiente documento presentamos un análisis sobre la infraestructura existente, donde es posible la implementación de plataformas para el despliegue de este tipo de aplicaciones, recabando las bondades y desventajas que cada una tiene.

Para esto se crearon diferentes casos de uso que representan posibles aplicaciones sobre esta metodología alternativa analizando su viabilidad. Entre ellos, se encuentran un sitio web estático, una enciclopedia colaborativa y una aplicación de comunicación en tiempo real.

6. Estado del Arte

En esta sección describiremos en qué se diferencian las aplicaciones descentralizadas de aquellas centralizadas, cuáles son las ventajas (y desventajas) del modelo de aplicación distribuido, y qué tecnologías existen actualmente para asistir en la creación de dichas aplicaciones.

6.1. Introducción a arquitecturas de red

Comunicar distintas computadoras es un trabajo que requiere coordinación por parte de todas las partes, protocolos para estandarizar la información que se transmite, e infraestructura para poder envíar cada bit de origen a destino. Entonces, se debe diseñar una red coordinada para poder ofrecer los distintos servicios a través de Internet. Para ello, existen dos arquitecturas principales.

Cliente-Servidor

Presente en la gran mayoría de las aplicaciones de Internet, el modelo *Cliente-Servidor* consiste en mantener un nodo central (servidor), quién se encarga de manejar la interacción entre los demás nodos (clientes), y entre clientes y el mismo servidor. Este modelo se clasifica como **centralizado**, debido a que la sub-red de sistemas depende del nodo servidor, y los clientes no tienen manera de comunicarse sin él ante una eventual caída del servidor.

Entre los servicios de Internet más utilizados que utilizan esta arquitectura se encuentra la World Wide Web, el servicio de e-mail (SMTP), el servicio de DNS, entre otros.

Peer-to-Peer

El modelo **peer-to-peer(P2P)** consiste en una red **descentralizada** que tiene distintos nodos (pares) capaces de comunicarse sin necesidad de un nodo central, por lo que se puede considerar que cada nodo cumple la función tanto de servidor como de cliente a la vez.

BitTorrent El servicio más utilizado que implementa este modelo es la red de BitTorrent, que implementa el protocolo del mismo nombre para compartir archivos entre pares. Esta red logra que el mismo nodo que descarga un contenido de la red sea a la vez el servidor para otro nodo que quiera acceder a ese contenido.

6.2. Diferencias y ventajas de cada arquitectura

Ambos modelos tienen ventajas y desventajas, y por lo tanto distintos casos de uso. El modelo cliente-servidor actualmente es la arquitectura mas utilizada,

Resiliencia Cuando un servidor se encuentra fuera de servicio, toda la red que depende de él no funcionará en tanto no se restaure el servidor. Para contrarrestar esta vulnerabilidad del modelo, se desarrollaron métodos a lo largo de los años. Una manera de evitar que la red se vuelva inoperativa es la de alojar diferentes instancias del servidor en diferentes zonas geográficas. Además, se pueden implementar medidas para evitar la caída del servidor, como los técnicas de balanceo de cargas y fuentes de energía alternativas para evitar eventuales cortes de electricidad.

No obstante, una red peer-to-peer puede ser incluso más robusta. Si hay suficientes pares en la misma y están lo suficientemente dispersos geográficamente, la desconexión de uno de ellos no desactiva toda la red. Esto permite que el modelo peer-to-peer pueda ser resistente a cortes de energía masivos y desastres naturales, lo que lo hace un modelo ideal para servicios prioritarios.

Cabe destacar que en una red de una cantidad limitada de pares, en donde no hay redundancia del contenido que se distribuye, es posible que al desconectar uno de los pares parte del contenido

se vuelva no disponible. Por lo tanto, si bien la red seguirá activa, no tendrá toda la funcionalidad que si puede ofrecer un servidor mientras siga en línea.

Escalabilidad La escalabilidad de una red peer-to-peer aumenta con la cantidad de nodos disponibles, dado que hay más recursos y, en una red bien diseñada, la carga se distribuye equitativamente. En un modelo cliente-servidor, garantizar escalabilidad se puede tornar costoso. Un servidor con mayor capacidad para comunicaciones entrantes y volumen de información tiene hardware de un costo mayor. En casos de aplicaciones de uso masivo puede ser necesario multiplicar la cantidad de nodos servidores para satisfacer la demanda de clientes.

Control del contenido Un servidor, al ser la pieza central de la red a la cuál pertenece, debe soportar múltiples conexiones en simultáneo. Para aplicaciones de alto tráfico, esto requiere una infraestructura que los usuarios suelen no poseer. Una solución es tercerizar el alojamiento de la aplicación servidor en plataformas de Cloud Hosting como pueden ser AWS, Azure, Google Cloud, entre otras. Estos servicios mantienen los servidores de numerosas aplicaciones de Internet, y por lo tanto, tienen la capacidad de modificar, censurar, o remover cualquiera de ellas si así lo desean.

Una red P2P no sufre de estos problemas, ya que por naturaleza los usuarios son quienes la alojan. Por lo tanto, remover contenido de ella resulta mucho mas complejo. Esto evita la censura en zonas donde el acceso a Internet es controlado y/o limitado, pero también puede incentivar a la distribución de contenido ilegal.

Seguridad La seguridad en las aplicaciones cliente-servidor se ha investigado por mucho más tiempo debido a la popularidad de este modelo. Además, al ser centralizado, el propietario del servidor puede bloquear conexiones y eliminar contenido malicioso de su plataforma de forma transparente para los usuarios.

El modelo descentralizado, en cambio, no cuenta con el desarrollo en términos de seguridad. En este caso, el cliente es el responsable de conectarse a redes de confianza, o de hacerlo mediante VPNs (Virtual Private Networks) para mantener el anonimato mientras se integre una red P2P. Sin embargo, cualquiera sea el modelo utilizado por una aplicación, la mayor parte de la seguridad dependerá de que tan segura sea la aplicación.

Persistencia Como varias otras propiedades del modelo cliente-servidor, depende de la integridad del servidor. Si el almacenamiento físico de este se ve afectado, los datos pueden perderse definitivamente. Por esta razón, es común tener un respaldo de los datos de la aplicación en otro disco u otro nodo para evitar la pérdida total de datos.

En una red descentralizada, la persistencia depende de la aplicación utilizada. En el caso de BitTorrent, cada nodo que se conecte y descargue un archivo, podrá compartir ese archivo con otros nodos, y por lo tanto ese archivo contará con una redundancia adicional, la cuál crece a medida que más personas descargan ese archivo. A pesar de esto, en casos donde el archivo es poco compartido, puede volverse inaccesible si los nodos que lo contienen se desconectan de la red.

Latencia Dada una conexión a Internet promedio, las velocidades manejadas por las aplicaciones cliente-servidor suelen ser aceptables. Sin embargo, en zonas en donde la conexión es escasa, o en casos en donde el servidor está lejos del cliente, la velocidad de transferencia de la aplicación puede verse afectada. Además, no es infrecuente encontrar cortes en videollamadas, juegos, y demás aplicaciones de tiempo real que siguen esta arquitectura. Como en la mayoría de defectos del modelo cliente-servidor, se puede solucionar agregando múltiples instancias del servidor. Por ejemplo, es común almacenar películas, videos y demás contenido de aplicaciones de streaming en distintos servidores de CDN (Content Delivery Network). Estas redes minimizan la distancia entre el usuario y el servidor, agilizando así la transferencia del contenido.

A pesar de los avances en la optimización del modelo cliente-servidor, las redes descentralizadas, cuando son eficientes y están bien pobladas, suelen ofrecer incluso mejores resultados. Esto se debe

a que la fuente de un contenido puede estar presente en múltiples nodos, lo que aumenta la probabilidad de que un nodo cercano tenga el contenido solicitado. La velocidad de transferencia que puede proporcionar un vecino con el contenido que requerimos generalmente superará la ofrecida por un servidor.

Costos Los costos de alojar una aplicación peer-to-peer suele ser nulo, ya que los mismos usuarios de ella son los encargados de proporcionar la infraestructura de la red.

Al contrario, alojar la aplicación en un servidor conlleva tener un servidor disponible, o bien contratar un servicio de web hosting, cuya tarifa suele aumentar a medida que la aplicación escala. En la mayoría de los casos, el costo termina siendo significativo, por lo que una aplicación descentralizada es una opción viable en escenarios en los que no se desee invertir mucho dinero.

6.3. Soluciones existentes

6.3.1. IPFS Deploy Action

Para desplegar un sitio web en IPFS, se puede utilizar un *GitHub Action* hecho por IPFS y lanzado en Febrero de 2025. Dicha herramienta es un script que se ejecuta con cada commit en un repositorio de GitHub, y permite compilar el sitio web, y luego alojarlo en IPFS utilizando un servicio de *pinning*, Filecoin, o bien un clúster de IPFS. Estas opciones se verán en detalle cuando se abarque la arquitectura de despliegue implementada, pero de todas ellas.

Sin embargo cuenta con una serie de desventajas. Por un lado, utilizando la opción de clúster sólo se puede instruir a un único nodo para que luego este suba el contenido al clúster. Un clüster es un conjunto de nodos orquestados para alojar el mismo contenido. Esto resulta en un *single point of failure* por lo que un clúster sólo podrá actualizarse si ese nodo está activo. Por otro lado, utilizar un GitHub Action nos ata necesariamente a GitHub, lo cuál supone otra limitación para quienes deseen optar por otra alternativa para su repositorio Git.

6.4. Ambientes y herramientas

Existen varios ecosistemas que apuntan a proveer un marco con el cuál desarrollar una aplicación descentralizada. A su vez, cada uno de ellos cuenta con herramientas especializadas para los diferentes tipos de aplicaciones.

6.4.1. IPFS

Un suite modular de protocolos para organizar y transferir datos, diseñado con los principios de *content addressing* (recuperación de archivos en base a contenido y no en base al nombre o id) y una red peer-to-peer. Su principal caso de uso es para publicar datos como archivos, directorios y páginas web descentralizadas.[10]

libp2p Colección de protocolos y utilidades para facilitar la implementación de una red peer-to-peer [60]. Entre sus herramientas, se encuentran diferentes mecanismos de seguridad, de transporte, y para descubrimiento de pares. Se creó con IPFS en mente, pero luego se expandió a un conjunto de protocolos independiente, el cual es utilizado por Ethereum actualmente. Los protocolos de interés para este proyecto son:

Protocolos de transporte Son los encargados de la comunicación entre nodos, de manera similar a la capa de transporte presente en toda red convencional. Se basan en tipos de transporte ya existentes, adaptados al uso peer-to-peer. Los protocolos principales son TCP, WebSockets y WebRTCDirect.

Protocolos de descubrimiento de peers Para encontrar un contenido en IPFS, se necesita saber la dirección del nodo que tiene dicho contenido. El principal protocolo para lograr esto se denomina Distributed Hash Table (DHT) [22] [71]. Es un registro clave-valor distribuido en todos los nodos que soporten este protocolo, que contiene la información necesaria para encontrar el contenido deseado. Cada nodo tiene una parte de esta tabla, y deberá preguntar a otros nodos hasta conseguir la dirección del nodo asociada a la dirección del contenido buscado.

Protocolos de mensaje

Kubo La implementación principal de IPFS es Kubo [58], una solución hecha en Go. Tiene su propio comando en la terminal, llamado `ipfs`, y también es utilizado en los demás front-ends de IPFS como IPFS desktop, la aplicación de escritorio de IPFS. Es la más madura y desarrollada de las dos implementaciones de IPFS, y cuenta con más funcionalidades.

IPFS Cluster Herramienta para orquestar distintos nodos de IPFS, con el objetivo de mantener disponible un contenido en la red de IPFS, aumentando su disponibilidad. Por defecto, cualquier nodo puede modificar la lista de archivos que mantiene el *cluster*.

Clusters colaborativos Son clusters que permiten que usuarios puedan colaborar con su nodo y aumentar la disponibilidad del contenido, sin tener permiso para modificar los archivos que se están manteniendo [50]. Esto es ideal para una aplicación comunitaria debido a esta medida de seguridad que permite al cluster ser apoyado por cualquier usuario de una comunidad.

OrbitDB Base de datos peer-to-peer descentralizada [69]. Utiliza por debajo IPFS para el almacenamiento de los datos, y *libp2p* como soporte para sincronizar cada base de datos entre todos los peers. Es *evetualmente consistente*, lo que significa que modificar una base de datos no asegura que este cambio esté disponible en el resto de los nodos bajo un tiempo específico. Esta herramienta tiene una implementación en Javascript, y otra mayormente abandonada en Go. Sin embargo, un nodo de OrbitDB puede ser utilizado en el entorno de un navegador con los parámetros correctos, lo cuál es útil para crear una aplicación web.

Helia Para crear una instancia de un nodo de OrbitDB se debe pasar como parámetro una instancia de Helia. Helia [34] es la implementación de IPFS para Javascript, y, como OrbitDB, también puede ejecutarse en un navegador. A diferencia de Kubo, Helia tiene un enfoque mucho más abierto, permitiendo la configuración del nodo libp2p utilizado internamente de manera abierta. Esto implica un trade-off claro: más configuración permite un nodo de IPFS que se puede adaptar a diferentes escenarios, pero también implica una mayor complejidad a la hora de lograr un funcionamiento correcto del nodo.

6.4.2. Blockchain

Tecnología basada en una cadena de bloques de operaciones descentralizada y pública. Esta tecnología genera una base de datos compartida a la que tienen acceso sus participantes, los cuáles pueden rastrear cada transacción que hayan realizado.[41]

6.4.3. Alternativas

Freenet/Hyphanet Programa de código abierto para compartir datos peer-to-peer con enfoque en la protección de la privacidad. Opera en una red descentralizada, promocionando la libertad de expresión facilitando la anonimidad de los datos compartidos y eludiendo la censura.[31][42]

7. Problema detectado y/o faltante

Los servicios actuales que proveen de infraestructura tienden a ser muy costosos para las pequeñas comunidades que necesitan tener un servicio con alta disponibilidad, accesible para todos y barato de escalar. Actualmente tampoco existe un estándar para aplicaciones cuyo stack sea completamente distribuido usando peer-to-peer.

7.1. Costos

La inversión para el mantenimiento de servidores puede ser un obstáculo a la hora de proveer una red a sus usuarios cuando se trata de una aplicación pequeña o startup. En estos casos, es común sacrificar la escalabilidad en pos de mantener los costos del alojamiento de la aplicación bajos.

En el otro extremo del espectro, se encuentran las aplicaciones en declive. Debido a la falta de incentivos financieros para justificar el mantenimiento, muchas veces la solución es desconectar los servidores definitivamente. Esto es especialmente frecuente en videojuegos multijugador que no cuentan con la capacidad de crear servidores dedicados. En tales situaciones, la empresa propietaria puede desactivar los servidores, lo que resulta en que el videojuego se vuelva parcialmente o completamente inutilizable. [12]

7.2. Interrupciones del servicio

Dada la naturaleza del modelo cliente-servidor, no es infrecuente que estas redes se vuelvan inaccesibles. Esto puede ocurrir deliberadamente por temas de mantenimiento, o accidentalmente debido a modificaciones en la aplicación del servidor durante el mantenimiento o actualización de la misma.

Uno de los episodios recientes de mayor revuelo ocurrió el 4 de Octubre de 2021, día en el que la familia de aplicaciones de Meta -Whatsapp, Facebook e Instagram -, estuvo fuera de servicio por seis horas. Esto ocasionó que mas de 3500 millones de usuarios se vieran afectados. En un articulo posterior, Meta reveló que la causa se debió a una falla en la herramienta de auditoría de los comandos que se envían a la red global de datacenters de Meta, la cual evitó que se pudiera detener el comando que desconectaba los servidores de la red. [57]

7.3. Zonas de censura

Es común que aplicaciones o sitios web comunitarios sean sometidos a su censura. La centralización de los servicios ayuda a que sea mucho más fácil bloquear su acceso, dado que es más fácil identificar y bloquear un único punto de acceso. En contraste, los sistemas descentralizados, suelen ser más resistentes porque no dependen de un servidor o servicio central que pueda ser intervenido fácilmente.

Uno de los casos más conocidos y vigentes es la censura de Wikipedia. Donde algunos gobiernos bloquean su acceso en un determinado lenguaje y otros en su totalidad. [13]

8. Solución implementada

Se implementaron tres casos de uso sobre los distintos ecosistemas a analizar. Se detalla el camino recorrido para llegar a cada implementación, haciendo uso de las distintas herramientas presentes en cada ecosistema para lograr cumplir con los requisitos funcionales presentados a continuación. Por último, presentamos un análisis cualitativo y cuantitativo de las ventajas y desventajas de cada caso dependiendo del ecosistema.

8.1. Casos de uso

8.1.1. Sitio web informativo

Este caso de uso consiste en desplegar un sitio o aplicación web, para poder recuperarlo utilizando un navegador. Es uno de los casos más sencillos y sirve como introducción para familiarizarse con cada ecosistema.

Requisitos funcionales

- **Landing page del proyecto**: sitio web informativo donde se presente información sobre este proyecto, los casos de uso y sus ecosistemas de despliegue.

8.1.2. Repositorio de conocimiento

Esta caso representa un repositorio con diferentes artículos, similar a *Wikipedia*. Es un servicio comunitario en donde se puede agregar información de distinta índole. Con este caso se analizó la capacidad de creación, modificación y recuperación de contenido por parte de los usuarios.

Requisitos funcionales

- **Edición:** los artículos dentro del repositorio deben poder editarse por cualquier persona que ingrese al sitio, y este cambio debe verse reflejado eventualmente en las demás personas que accedan a ese artículo.
- **Historial de versiones:** cada artículo debe tener una lista de versiones anteriores, junto con hipervínculos con los cuáles acceder a ellas.
- **Búsqueda:** una persona debe poder realizar una búsqueda global que incluya todos los artículos.

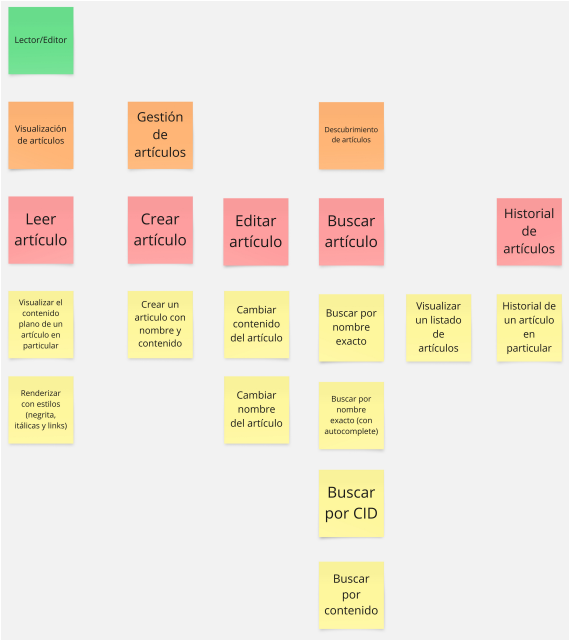

Figura 1: *User Story Map* del repositorio de conocimiento

Package Este caso de uso se implementó en los packages **Astrawiki** [6] (implementación sobre el ecosistema IPFS) y **Astrawiki-eth** [5] (implementación sobre el ecosistema Blockchain). En las subsiguientes secciones, las menciones a Astrawiki referirán a este caso de uso.

8.1.3. Mensajero en tiempo real

Este caso se enfoca en la capacidad de la infraestructura de enfrentarse a situaciones de *tiempo real* como puede ser un chat de texto o de audio. En particular, nos centramos en el caso de chats de texto para un grupo de usuarios en donde los mensajes sean públicos.

Requisitos funcionales

- **Usuarios:** se deben contar con usuarios que puedan iniciar sesión con una clave.
- **Grupos públicos:** grupos de chat de texto, donde cualquier usuario puede ingresar y ver los mensajes del resto, así como también participar enviando sus propios mensajes.
- **Respuestas:** un usuario debe poder responder mensajes anteriores dentro de un mismo chat.

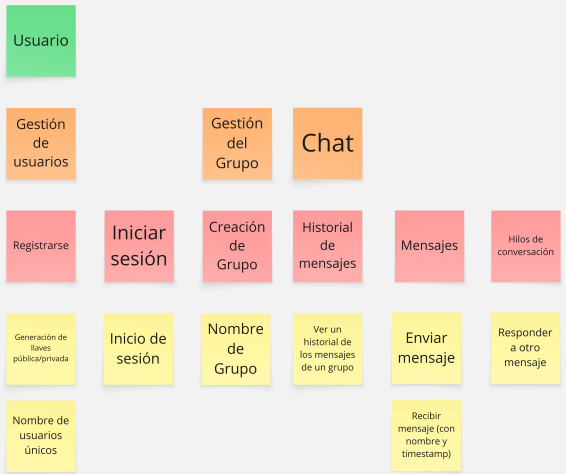

Figura 2: *User Story Map* del mensajero en tiempo real

Package Este caso de uso se implementó en los *packages* **Astrachat**[4] (implementación sobre el ecosistema IPFS) y **Astrachat-eth**[3] (implementación sobre el ecosistema Blockchain). En las subsiguientes secciones, las menciones a Astrachat referirán a este caso de uso.

8.2. Proceso de descubrimiento

Durante la implementación de los casos de uso para cada ecosistema, se fue desarrollando un mejor entendimiento de lo que se estaba creando. Logrando así generar distintas abstracciones que representan la infraestructura general para la implementación de los casos de uso.

Para cada ecosistema hay 2 principales infraestructuras en acción. La infraestructura de despliegue y la infraestructura de aplicación.

La **infraestructura de despliegue** es la encargada del *hosting* de una aplicación web. Con esta se distribuye y permite el acceso a lo que es el *front-end* de las aplicaciones, como también el código para el funcionamiento de la aplicación, si se trata de una aplicación no estática. Cualquier aplicación que desee ser accedida por la web va a hacer uso de esta infraestructura, como es el caso del sitio web informativo.

La **infraestructura de aplicación** es la encargada de la lógica de la aplicación, como es el almacenamiento de datos, como la conexión entre pares. Aplicaciones que requieran de mantener un estado y permitir la modificación de parte de los usuarios van a necesitar hacer uso de esta infraestructura, como es el caso del repositorio de conocimiento y mensajero en tiempo real.

Al lograr encontrar estas abstracciones, se permite que generar nuevos casos de uso sea mucho más fácil, ya que la mayoría de la lógica sobre el el ecosistema se encuentra encapsulada dentro de ellas, logrando que el desarrollo de un nuevo caso de uso se concentre únicamente en sus requisitos y no en el ecosistema en el que se encuentra.

A continuación se va a explicar como se componen y funcionan ambas infraestructuras en cada ecosistema.

8.3. IPFS

8.3.1. Infraestructura de despliegue

En IPFS, es posible desplegar una aplicación web subiendo un directorio con todos los archivos estáticos necesarios para el funcionamiento en un navegador, incluyendo el código necesario a nivel aplicación. Esto se puede realizar manualmente mediante cualquier cliente de IPFS, como Kubo[58] o Helia[34], y devuelve un *content identifier* (CID) [51] que representa esa versión de la aplicación.

Cualquier usuario puede publicar su sitio web en la red de IPFS a través de un nodo local de manera gratuita y poco tiempo. IPFS provee un tutorial en su página de cómo realizarlo [36]. A continuación se detallará las implicaciones que tiene desplegar una aplicación web de esta manera, y que alternativas existen para publicar en IPFS.

Al subir un archivo —por ejemplo, código HTML— su contenido se inserta en una función de hash, y así se obtiene su CID. Desde ese momento, cualquier nodo que desee obtener el archivo puede encontrarlo utilizando dicho CID. Sin embargo, no se asegura la persistencia del archivo, y dejará de ser accesible luego de un tiempo. Esto se debe al *garbage collector* [52] implementado por IPFS, que desecha datos para liberar almacenamiento de forma arbitraria. Por esta razón existe el concepto de [74]: *"pinear"* o fijar un archivo o directorio significa instruir al nodo IPFS para que trate dicha información como esencial y, por lo tanto, no lo descarte.

No obstante, fijar un archivo o directorio no asegura su disponibilidad indefinida en el tiempo, ya a que esta depende de que el nodo que que fije el contenido esté activo, o de que otros nodos que hayan accedido al archivo y aún lo tengan en su caché. Para mejorar la disponibilidad de un archivo, lo ideal es que varios nodos fijen el contenido, de modo que otro nodo que desee obtener el contenido pueda hacerlo desde cualquiera de ellos.

Para lograr que el contenido persista en la red sin necesidad de que el nodo local esté activo, existen opciones para delegar el pinning del archivo o directorio. Existen servicios de pinning, el servicio ofrecido por Filecoin, y clústeres colaborativos, que actúan fijando los archivos en múltiples nodos, aumentando no solo su disponibilidad sino también su distribución, y por ende logrando un acceso más rápido al contenido.

Servicios de *pinning* La manera más fácil de asegurarse que los datos estén disponibles y se persistan es usar un servicio de *pinning* [54]. Estos servicios cuentan con varios nodos que fijan archivos. De esta manera, ya no es necesario contar con un nodo local que los aloje. Algunos ejemplos de servicios de pinning incluyen Fleek [30], Filebase[29] y Pinata [73].

Estos servicios no van de la mano con la filosofía de aplicaciones estrictamente comunitarias. Por un lado, los servicios de *pinning* tienen un modelo gratis con funcionalidad limitada o capacidad de almacenamiento limitado. Por otro lado, se depende de estos servicios, lo que en esencia centraliza el proceso de despliegue de la aplicación o sitio web. Si por algún motivo el servicio dejara de fijar los archivos, estos pueden dejar de estar disponibles en la red IPFS, e incluso pueden perderse por completo. Esto rompe completamente con la naturaleza de aplicaciones descentralizadas y pasa a tener una centralización tercerizada similar a utilizar un *cloud hosting*.

Filecoin Filecoin [59] es una Blockchain creada por IPFS que ofrece una red en la cuál un nodo puede proporcionar almacenamiento a cambio de un incentivo económico en forma de criptomoneda. Utilizando esta alternativa, un cliente puede ofrecer FIL, la criptomoneda de Filecoin a cambio de asegurar que el contenido está alojado en un nodo proveedor de almacenamiento. Este nodo proveedor debe proporcionar diariamente prueba de que el contenido indicado está disponible.

Es distinto de los servicios de pinning vistos, ya que los nodos que proveen almacenamiento dentro de Filecoin no pertenecen necesariamente a una misma organización, sino que se distribuyen a lo largo de su red. Sin embargo, requiere una inversión constante para mantener la persistencia del contenido, y por lo tanto no es ideal como solución de almacenamiento permanente para una aplicación comunitaria.

Clústeres colaborativos Un *clúster* es un grupo de nodos de IPFS que actúan en conjunto para fijar un contenido. Funcionan sincronizando su *pin set*, o sea, su lista de archivos y directorios fijados en un momento dado. Un clúster *colaborativo* sigue esta premisa, pero permite que los usuarios puedan colaborar con su nodo local para el pinning de la aplicación sin tener la posibilidad de modificar los archivos, la cuál es delegada a nodos especiales que tienen la capacidad de orquestar el clúster en conjunto. Así, se logra que la misma comunidad mantenga en servicio el mecanismo de despliegue de la aplicación, lo cuál es acorde a la filosofía de aplicaciones comunitarias.

Actualmente, esta alternativa es poco explorada, por lo tanto no existe una forma fácil de creación, seguimiento y descubrimiento de estos clústeres. IPFS cuenta con una página con clústeres conocidos con los cuales se puede colaborar [50], pero la cantidad es limitada.

Por otro lado, el principal problema es que los clústeres obligan a los nodos a fijar la totalidad de sus archivos, lo cuál puede significar un uso excesivo de almacenamiento necesario para colaborar. Hacer *sharding* sobre el pin set, o sea, fijar parte del contenido de un clúster, es posible utilizando los parámetros de `replicator_min` y `replicator_max` al agregar un pin, que fijan un límite mínimo y máximo sobre la cantidad de nodos que tienen ese pin. Sin embargo, no es recomendado para clústeres colaborativos debido a la falta de *proof of storage* [38] [16]. Esto se debe a que, debido a la manera en la que fue diseñada la arquitectura de IPFS, un nodo no confiable puede falsificar la lista de archivos que está fijando, por lo que hay una posibilidad de que una parte del contenido no esté en ninguno de los nodos, y por ende el contenido esté incompleto.

Acceso y mutabilidad Para buscar un contenido, un nodo de IPFS realiza una búsqueda a través de su CID, el cual es único. Debido a que es único, el CID cambiará si el contenido del sitio web o aplicación web cambia, ya que el contenido será distinto. Esto vuelve el proceso de despliegue altamente impráctico, ya que se necesitaría compartir un nuevo CID cada vez que se actualice una página.

Este problema puede ser resuelto con la ayuda de *punteros mutables*. Estos punteros son un objeto de IPFS que apunta a un CID determinado, previamente elegido por el usuario. El CID al que apunta el puntero puede ser cambiado, por lo tanto permiten compartir la dirección del puntero una única vez y actualizar el CID al cuál apunta cada vez que se haga un cambio.

IPNS InterPlanetary Name System (IPNS) [49] es un sistema que permite crear punteros mutables y obtener su dirección en forma de CIDs conocidos como *names* o *nombres de IPNS*. Estos nombres de IPNS pueden considerarse como enlaces que pueden actualizarse, conservando al mismo tiempo la verificabilidad del content addressing.

Un nombre de IPNS es un hash de una [2] clave pública. Está asociado a un *IPNS record* [56] que contiene la ruta a la que se vincula, entre otra información. El titular de la clave privada puede firmar y publicar nuevos registros en cualquier momento.

Es posible utilizar IPNS con uno de estos posibles enfoques:

- **Consistencia:** garantizar que los usuarios siempre resuelvan el último registro de IPNS publicado, a riesgo de no poder resolverlo.

- **Disponibilidad:** resolver un registro de IPNS válido, a costa de potencialmente resolver un registro desactualizado -o sea, con un CID previo.

El registro IPNS se encuentra a través de la **Distributed Hash Table** (DHT) [22]. Todos los nodos de IPFS participan alojando el contenido de la DHT de forma colaborativa. Por lo tanto, la DHT actúa como un "directorio"descentralizado, donde la clave pública es un identificador. Esta tabla ayuda a localizar el registro IPNS que apunta al contenido deseado, entre otras funciones. Para entender mejor cómo IPNS funciona se puede consultar la documentación de IPFS.

IPNS es una buena forma de obtener mutabilidad dentro de IPFS. Una vez que se aloja un contenido en IPFS y se apunta a él mediante un *nombre* de IPNS, el mayor problema pasa a ser la manera de acceder a IPNS en sí. El hecho de que los nombres sean hashes alfanuméricos, y no nombres legibles o memorables para humanos, representa una dificultad adicional a la hora de alojar un sitio web al cuál los usuarios puedan acceder fácilmente. A continuación se analizará dos alternativas para solucionar este problema.

```
/ipns/k51qzi5uqu5dhkdbjdsauuyk5iyq82uzpjb0is3x6oy9dcmmr8dbcezv7v9fya
```

Figura 3: Ejemplo de la dirección de un nombre de IPNS

DNSLink IPNS no es la única forma de crear punteros mutables en IPFS. DNSLink [24] utiliza registros *DNS TXT* para asignar un nombre DNS a una dirección IPFS como un CID o un nombre de IPNS. Se puede usarlos para que siempre apunten a la última versión de un objeto en IPFS.

DNSLink actualmente es más rápido que IPNS, utiliza nombres legibles por humanos y también puede apuntar a nombres de IPNS. A pesar de ello, tiene un problema muy fundamental y es que se utiliza el protocolo **DNS**, el cual tiene claras deficiencias con la filosofía de aplicaciones comunitarias.

La más importante es que, aunque DNS tenga claras ventajas, como ser un sistema distribuido y escalable, es también un sistema algo centralizado. Las autoridades centrales como *ICANN* gestionan las raíces del DNS. Esto hace que un registro DNS sea fácil de censurar, a nivel de registrador como también a nivel *ISPs*.

ENS **Ethereum Name Service (ENS)** [26], es el protocolo de nombres descentralizado que se basa en blockchain *Ethereum*. Funciona de manera similar a DNS, en el sentido de que los nombres ENS resuelven a nombres legibles para humanos. Como esto se computa en la blockchain de Ethereum, es seguro, descentralizado y transparente. Está diseñado específicamente para traducir identificadores como direcciones de billeteras de criptomonedas, hashes, metadata, entre otros, incluyendo direcciones de IPFS.

Es posible configurar un registro ENS para que se resuelva automáticamente la dirección IPNS, proporcionando nombres legibles para humanos que son más fáciles de compartir y acceder, y solucionando el principal problema de IPNS hasta este punto. Además, cuando se quiera actualizar el contenido, no será necesario modificar el registro ENS en sí, ya que siempre se va a apuntar al mismo nombre de IPNS.

Cabe aclarar que adquirir un dominio ENS tiene un costo, que depende de varios factores [28]:

- El largo del nombre. Un nombre con menos caracteres tiende a tener un valor mayor.
- Cuán reciente expiró la licencia del dominio. Si un dominio expiró recientemente, se le aplica un precio *premium* que decrece con el tiempo. Un dominio con mayor uso aumenta su precio.
- El valor del gas actual, que depende de la congestión de la blockchain.

Acceso desde un navegador Por último, se necesita una manera de acceder a los archivos alojados en IPFS. En navegadores que soportan IPFS y ENS —como Opera [9] y previamente Brave [11]— se puede acceder directamente. En la mayoría de los navegadores, sin embargo, esta no es una opción. Para lograr un mayor alcance que incluya estos navegadores, se requiere el uso de un *Ethereum gateway* que entienda un dominio ETH, y un *IPFS gateway* para lograr obtener el contenido de IPFS [53].

Un IPFS gateway es un nodo que recibe requests HTTP que contienen una dirección de IPFS, busca el contenido en la red de IPFS, y lo devuelve en una HTTP response. Esto es útil tanto para archivos como para directorios. Algunas gateways tienen la funcionalidad de mostrar una página web de manera correcta cuando un directorio tiene la estructura indicada. Esto nos es particularmente útil para poder mostrar una página moderna de la misma manera que se haría utilizando un servidor HTTP.

Una lista de gateways disponibles puede obtenerse utilizando el Public Gateway Checker [76] proporcionado por IPFS.

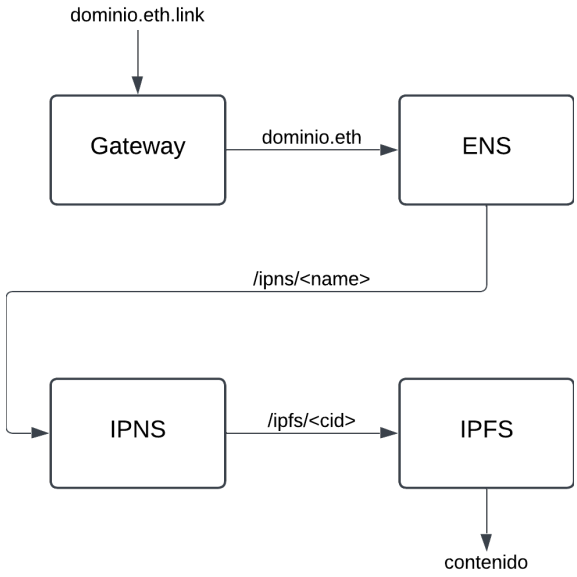

Figura 4: Mapa de la traducción de un dominio al contenido de IPFS

En nuestro caso, se utilizó el servicio de Limo [61], que soporta direcciones de ENS, y también recupera archivos de IPFS por lo que actúa como una gateway de ambos servicios. Para utilizarlo, simplemente se requiere agregar un sufijo `.link` al nombre de ENS. Por ejemplo, `dominio.eth` puede ser accedido mediante `dominio.eth.link`. A su vez, resuelve `dominio.eth` a una dirección de IPFS, y por lo tanto obtiene los archivos de esa dirección automáticamente.

Despliegue continuo En un proyecto de aplicación web centralizada, es común automatizar el proceso de despliegue con cada cambio que se realiza. Normalmente este proceso se activa con cada nuevo commit en una rama de Git especifica, e incluye todas las etapas necesarias para convertir el contenido de un repositorio Git en código estático listo para ser desplegado. También puede incluir más pasos que incluyan actualizaciones en el back-end.

Yendo al caso específico de aplicaciones web comunitarias, el script debe ser ejecutado en los nodos confiables, ya que una *Github action* no puede utilizar un nodo IPFS que requiera puertos abiertos. En este tipo de aplicaciones, al tener una jerarquía mayormente horizontal, no hay un servidor central que orqueste esta actualización, sino que se necesita que cualquier nodo confiable pueda actualizar su contenido e instruir a los nodos colaborativos para actualizar su contenido de igual forma. Todo esto debe ser posible incluso cuando los nodos no reciben la actualización al mismo tiempo, es decir, no debe haber *race conditions*.

Una forma de lograr esto es, por ejemplo, utilizar un algoritmo de elección de líder u otro

algoritmo distribuido para elegir el nodo responsable de indicar al resto de los nodos el nuevo contenido a fijar. Sin embargo, esta manera de realizar la actualización implica una capa adicional de complejidad que no es necesaria debido a la naturaleza de IPFS.

Como ya se ha mencionado, si dos nodos suben el mismo contenido, obtendrán el mismo CID. Esto puede ser utilizado para que cualquier nodo confiable pueda actualizar el contenido y el nombre de IPNS independientemente del resto de los nodos confiables. Cuando se detecte un cambio nuevo, el nodo puede obtener el código estático, y acto posterior, indicar al resto de los nodos del clúster que fijen el CID especifico. En el caso de que sea el primer nodo en detectar el cambio, deberá instruir al resto del clúster para que dejen de fijar el CID antiguo. En el caso en que otro nodo haya detectado la actualización antes, no deberá actualizar ningún pin del clúster debido a que el mismo CID ya va a estar presente en la lista de pins.

Compilación Las herramientas de compilado no siempre son deterministas en los archivos compilados que genera. `Next.js`, por ejemplo, genera diferentes archivos estáticos en dos compilaciones basadas en el mismo código fuente. Esto es un problema para el enfoque propuesto, debido a que si dos nodos compilan el mismo código, el CID puede ser diferente. Para mitigar esto, se decidió hacer uso de un *hook* que compile el código con cada *commit* en la rama principal una única vez por cambio realizado. De esta manera, los nodos confiables pueden detectar el cambio en la rama utilizada para alojar los archivos estáticos, y hacer *pull* sobre esos archivos y, por lo tanto, obtener un mismo CID.

Jerarquía En base a este análisis, podemos concluir que la mejor forma de desplegar una página web estática en IPFS es a través del uso de un clúster colaborativo compuesto por nodos que se integren con el proyecto de Git dado, así como una dirección IPNS a la cuál actualizar cada vez que hay un cambio, y un registro ENS para traducir la dirección IPNS a un nombre legible.

Como el nombre de IPNS cambiará a lo largo del tiempo en tanto se realicen cambio en el proyecto, se vuelve necesario seleccionar un grupo de nodos que se les confíe con tal fin. Esto se debe a que, de lo contrario, un posible atacante podría modificar el registro para invalidarlo o cambiar el contenido al que apunta. Por la misma razón, no cualquier nodo dentro del clúster debe ser capaz de cambiar el *pin set*, o lista de CIDs fijados por el clúster.

IPFS Cluster tiene en cuenta esto, y hace la distinción entre un nodo *trusted* y un nodo *follower* para su implementación de clústeres *colaborativos*[15]. Para esta herramienta, se utiliza las denominaciones de nodo confiable y nodo colaborador, respectivamente.

Nodo confiable Este nodo tiene la capacidad de modificar el nombre IPNS, como también actualizar la configuración del mismo, y el *pin set*. Son una parte esencial del clúster, ya que sin estos nodos no se podrá modificar el contenido. Esto no supone una desventaja ni tampoco hace que la solución se vuelva centralizada en el grupo de nodos confiables actual, debido a que los usuarios de la comunidad pueden crear su propio grupo de nodos confiables y actualizar el contenido por su cuenta, similar a realizar un *fork* en un proyecto de Github.

Nodo colaborador Únicamente se encarga de fijar los archivos establecidos por los nodos confiables, y actualizar su *pin set* cuando se lo indique. Al igual que los nodos confiables, debe fijar la totalidad de los archivos. Su finalidad es aumentar la disponibilidad del contenido y evitar que la información se pierda.

En un escenario ideal, existen varios nodos confiables disponibles en simultáneo. Esto previene un posible *single point of failure* y asegura que el clúster siempre se encuentre en un estado válido.

Configuración Para que un usuario pueda conectarse y contribuir como colaborador a un clúster, la herramienta de terminal `ipfs-cluster-follow` [55] requiere una dirección de IPFS de la cuál obtener el archivo `service.json` [17]. Este archivo de configuración contiene todos los datos necesarios para que un colaborador pueda unirse. Además, está sujeto a modificaciones, debido a que el

archivo contiene las *multiaddresses* [65] de cada nodo confiable en forma de lista, por lo que agregar o remover un nodo confiable implica modificar el archivo. Es por esto que el proceso de despliegue también debe incluir este archivo. Desde la detección de una actualización en un repositorio de Git que lo contenga, el fijado del nuevo `service.json` al clúster, hasta la actualización de un nombre de IPNS que pueda distribuirse a los usuarios que quieran colaborar.

```
/ip4/123.123.123.123/udp/9096/quic/p2p/12D3KooWLw...yPcuZJR
```

Figura 5: Ejemplo de una *multiadress* posible que utiliza el protocolo QUIC.

Limitaciones Este enfoque, a cambio de ofrecer una solución comunitaria y descentralizada, tiene desventajas o aspectos a mejorar:

Necesidad de tener nodos confiables Estos nodos van a ser los encargados de administrar el clúster, y actualizar el IPNS. La distinción entre nodos confiables y nodos colaborativos es necesaria para evitar que un potencial atacante pueda modificar el CID al que apunta el nombre de IPNS, o modificar el contenido que fija el clúster colaborativo.

Actualización del contenido Por cada cambio que se realice en el directorio de la página, se deberá fijar el nuevo contenido al clúster, y por lo tanto todos los colaboradores tendrán que obtener todo el directorio nuevamente. Esto puede claramente volverse costoso con contenido de tamaño considerable.

Cache de IPNS El parámetro TTL de IPNS indica cuanto 'vive' un valor asociado a un nombre de IPNS en la cache de un nodo antes de forzar a este a volver a buscar el valor en la DHT. El problema que tiene esto es que, si se pone un valor muy elevado, un nodo gateway no buscará la actualización hasta que se cumpla el periodo y por lo tanto el registro de IPNS no se actualizará. Por otro lado, si se elige un valor muy corto, siempre se buscará el valor en la DHT, generando latencia al no utilizar el cache disponible. Pero a su vez, el nombre de IPNS en un nodo siempre tendrá la última versión que encuentre.

Claves privadas compartidas Cómo la actualización de un nombre de IPNS está firmada con una clave privada, todos los nodos confiables deberán tener la misma clave para poder potencialmente actualizar el registro IPNS y así evitar tener un único nodo con esa responsabilidad. Esto elimina un punto de falla único, pero aumenta las chances de que esa clave privada llegue a manos de un posible atacante.

Apertura de puertos IPFS Cluster utiliza el puerto 9096 para la comunicación entre nodos, el cual se tiene que abrir para un correcto funcionamiento. Esto puede suponer un esfuerzo adicional para usuarios que deseen colaborar.

Implementación Una vez explicado el análisis inicial y las decisiones que se tomaron para poder lograr un servicio que automatice y facilite parte del despliegue de una aplicación web, se detallará la solución realizada para el nodo confiable. El resultado es una herramienta que se puede levantar utilizando un comando, y automáticamente publique el contenido ubicado en el repositorio de Git dado, encargándose de mantenerlo disponible, de orquestar el *pin set* del clúster, y detectar cambios. El repositorio se puede encontrar en el repositorio de GitHub [79].

Arquitectura general La herramienta está compuesta por tres contenedores:

- **Kubo:** el nodo de IPFS encargado de conectarse a la red de IPFS para publicar y obtener el contenido necesario.

- **IPFS Clusters:** gestiona el contenido fijado y coordina con otros nodos del clúster.
- **Watcher:** observa los repositorios de Git del proyecto y del archivo `service.json`, y orquesta acciones en los otros dos contenedores.

Todos los contenedores están orquestados mediante Docker Compose. El contenedor watcher está basado en Alpine Linux y utiliza scripts de shell portables. La comunicación entre contenedores se realiza mediante sus respectivas APIs HTTP [40] [39].

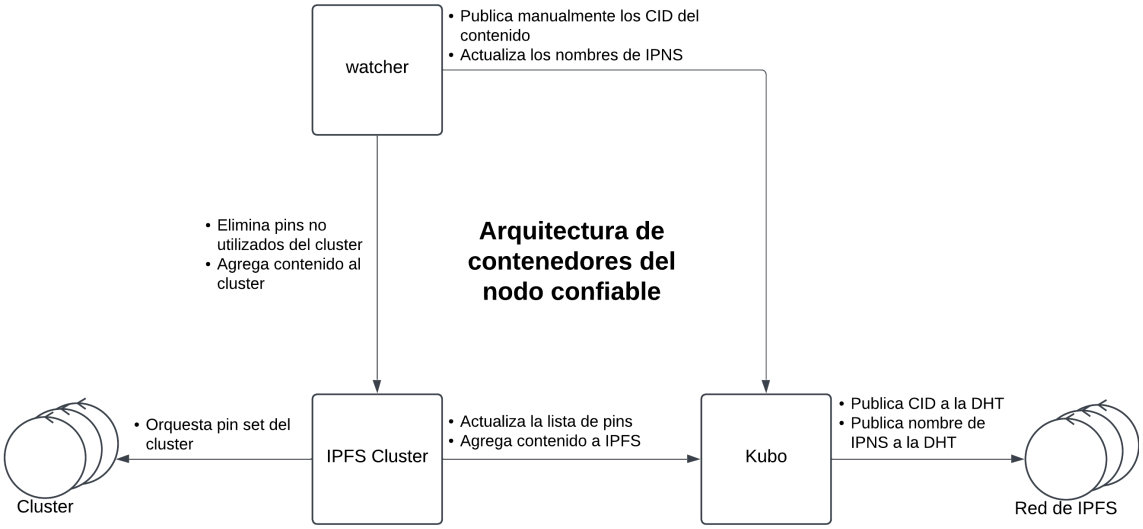

Figura 6: Mapa de interacciones entre los contenedores del nodo confiable

Funcionamiento del Watcher Este módulo del nodo confiable utiliza Git para comparar el último commit de la rama remota contra una copia local que se clona cada vez que se inicia. De esta manera, puede detectar cuando un nuevo cambio ocurre (tanto en el contenido como en `service.json`), e iniciar el proceso para obtener el nuevo cambio y desplegarlo. Dicho proceso se compone de los siguientes pasos:

1. Subir el contenido y el `service.json` al Cluster, y obtener ambos CIDs.
2. En base a los CIDs obtenidos, publicar ambos manualmente utilizando Kubo.
3. Esperar a que todos los nodos dentro del clúster hayan fijado los nuevos CIDs.
4. Actualizar los dos nombres de IPNS para que apunten a los nuevos CIDs.
5. Eliminar los pins antiguos del clúster.

Disponibilidad En el proceso mencionado para desplegar los cambios, existen dos factores que pueden afectar a la disponibilidad del contenido luego de recibir una actualización.

Por un lado, el nombre de IPNS puede no haberse actualizado en todos los nodos de la DHT, lo que provoca que algunos nodos apunten a la versión anterior del contenido. Esto se soluciona asegurándose de publicar el nuevo valor del nombre de IPNS **antes de instruir al clúster** para que deje de fijar la versión anterior.

Por otro lado, el contenido nuevo puede no estar disponible inmediatamente, ya que la publicación del CID en la DHT por parte del clúster se realiza de manera asíncrona. Para solucionar esto, se optó por publicar manualmente el CID con Kubo, de forma secuencial, antes de actualizar el nombre de IPNS. La desventaja de este enfoque es el tiempo adicional requerido para publicar el contenido, a cambio de garantizar su disponibilidad en todo momento, ya sea en su versión actual o en la nueva.

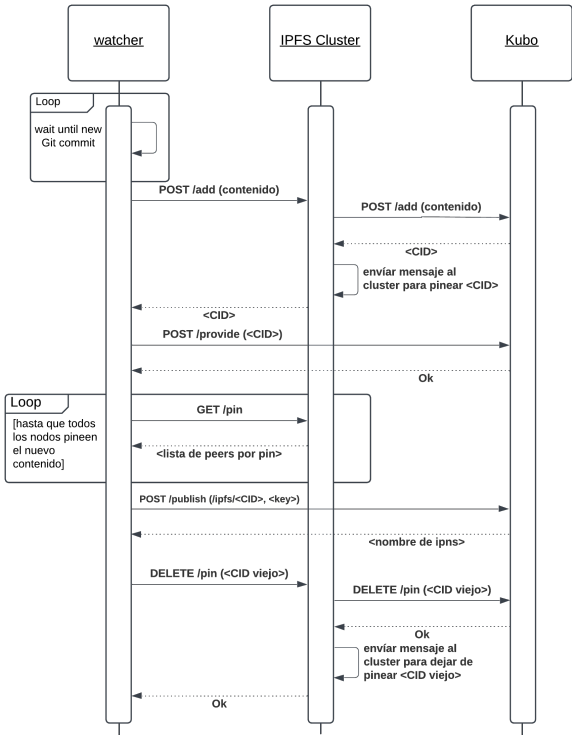

Figura 7: Diagrama de secuencia para el caso en que watcher detecta un cambio. Notar que para mayor claridad se omite los pasos para desplegar el nombre de IPNS del `service.json`, al ser exactamente los mismos que en el caso de un contenido.

Persistencia de la identidad del nodo Sabiendo que para ser un nodo confiable se debe tener su multiaddress en el archivo de `service.json`, es conveniente mantener el mismo PeerID a lo largo del tiempo y en distintas ejecuciones de la herramienta. Para ello, se debe indicar una *identidad* que consiste de un PeerID, y una clave privada. Esto asegura que el nodo siempre se inicie con la misma identificación.

Gestión de claves de IPNS Debido a la naturaleza de IPNS, un nombre solo puede ser modificado por un nodo que posea una clave privada determinada. Por ello, todos los nodos confiables deben tener las mismas clave privada de IPNS, una para el contenido y otra para `service.json`.

Para facilitar la inicialización, la herramienta provee un script que ayuda a generar la configuración y obtener los parámetros necesarios paso a paso. Esto incluye una identificación para el nodo, claves para IPNS, las direcciones de los repositorios de Git, y la IP pública necesaria para conectar los nodos a la red de IPFS.

Integración con Git La manera en la que el contenedor *watcher* puede detectar un cambio en el repositorio es consultando el repositorio remoto de Git cada minuto para identificar un cambio realizado y accionar el script de despliegue. Se requiere que el repositorio del contenido sea público, ya que la identificación por SSH o usuario y clave no están disponibles fácilmente dentro de un contenedor. De todas maneras, el contenido o archivos estáticos en el caso de una aplicación web ya son públicos por naturaleza, y debido al enfoque comunitario dado, que un repositorio necesite ser público no representa una restricción apreciable.

Resultado La solución implementada logra automatizar el despliegue y la publicación de contenido en IPFS de forma confiable, simplificando muchos aspectos de IPFS y los clústeres colaborativos. Mediante un comando `make up` se levanta un nodo confiable que automáticamente puede desplegar y mantener actualizado el contenido que se desee. Cabe destacar que, si bien el

enfoque está diseñado para aplicaciones web, esta herramienta permite el despliegue de cualquier tipo de contenido, como repositorios, documentación, etcétera.

Combinando esta herramienta junto con un dominio ENS y un gateway con el cuál acceder al contenido, se obtiene una aplicación web cuyo uso es equiparable a la de un servidor HTTP moderno, sin diferencias perceptibles para el usuario, y de manera comunitaria, descentralizada, y económica.

8.3.2. Infraestructura de aplicación

Para aplicaciones que requieran mantener un estado y permitir que usuarios puedan modificarlo, no es suficiente con la infraestructura que explicamos anteriormente, ya que no hay una noción de estado y solo se le permite cambiar su contenido a los dueños de lo que se despliega. Es por eso que es necesaria otra infraestructura, la cual nos provea de esas necesidades.

Esta infraestructura surgió en base a un extenso desarrollo del cual nos ayudó a entender y encontrar la abstracción de lo que se estaba creando. Al principio del desarrollo, esta era gran parte de la arquitectura del primer caso de uso no estático que realizamos, el repositorio de conocimiento, para luego convertirse en una implementación propia, la cual llamamos **AstraDB**.

A continuación pasaremos a explicar cómo es la arquitectura que compone a AstraDB, cómo fue su evolución y que decisiones se tomaron a lo largo de su desarrollo, como también cómo podemos hacer uso de ella para fácilmente crear las aplicación que venimos a analizar, aplicaciones comunitarias, distribuidas y descentralizadas dentro del ecosistema de IPFS, tal como lo son el repositorio de conocimiento y el mensajero en tiempo real, las cuales hacen uso de AstraDB para su funcionamiento.

Etapa de investigación Al comenzar con el desarrollo del repositorio de conocimiento nos encontramos con un desafío, cómo podemos lograr que una aplicación dentro del ecosistema de IPFS pueda tener, modificar y guardar un estado.

Dada la naturaleza de IPFS, como explicamos anteriormente, no está pensado para alojar cambios en tiempo real. El modelo de direccionamiento por contenido implica que cualquier modificación genera un nuevo identificador (CID), lo que resulta inconveniente para actualizar un recurso directamente sin mecanismos adicionales. Por esta razón, utilizar únicamente el conjunto de protocolos que IPFS ofrece no nos resulta conveniente para aplicaciones dinámicas.

Como sucede con un caso de uso muy similar al repositorio de conocimiento que queremos implementar, el proyecto de **Distributed Wikipedia Mirror**[23], el cual consistió en poner una versión de wikipedia en IPFS, únicamente funciona como versión Read-Only, y con snapshots manuales, lo cual es totalmente posible de hacer con la infraestructura que explicamos anteriormente.

Es por esto que resulta de un verdadero desafío lograr que una versión Read-Write sea posible sin sacrificar los principios de decentralización que IPFS nos provee. Para abordar esta limitación nos llevó a buscar herramientas complementarias dentro del ecosistema y ahí fue cuando nos encontramos con **OrbitDB**[69].

Representación de los datos OrbitDB es una base de datos peer-to-peer, distribuida y sin un servidor central. Utiliza IPFS para el almacenamiento de datos y **Libp2p**[60] para sincronizar automáticamente las bases de datos con otros peers. Es una base de datos **eventualmente consistente** que utiliza Merkle-CRDTs para escrituras y fusiones de base de datos libres de conflictos, lo que hace que OrbitDB sea una excelente opción para aplicaciones p2p y descentralizadas.

OrbitDB ofrece varios tipos de bases de datos para diferentes modelos de datos y casos de uso. Algunos que se asemejan más a bases de datos convencionales, como puede ser un simple key-value y otros más distintos como puede ser uno de eventos secuenciales.

Uno de los requisitos de la infraestructura desarrollada es lograr una verdadera descentralización, significando que no haya una entidad o persona con mayores permisos sobre el resto, esto se traduce, en parte, a permitir que cualquiera que quiera pueda crear y/o modificar la base de

datos sin ninguna restricción. Para lograr esto, OrbitDB nos permite indicar que cualquier nodo tenga permiso de edición sobre una base de datos. El problema es que esto también permitiría que cualquier usuario pueda eliminar información de la base de datos y es algo que no nos podemos permitir y es esta la principal razón por la cual no podemos utilizar un tipo como key-value, o documentos y necesitamos otro que se adecúe más a nuestro caso.

OrbitDB nos provee de otro tipo de base de datos, el tipo de events. Un tipo de base de datos el cual es inmutable (append-only), un log transversal que muestra un historial que se puede recorrer. Sin embargo surgen nuevos desafíos.

Gracias a que es un tipo append-only ya no se tiene el problema de posible perdida de información, ya que un usuario solo puede agregar un nuevo valor a la base de datos y no eliminar valores agregados previamente, sin embargo nos surge otro problema, como representamos la información con una base de datos que solo se puede agregar?

La solución es separarnos de la idea de que se tiene que tener una única base de datos que albergue toda la información. Al pensarlo desde el punto de vista del repositorio de reconocimiento, podemos representar cada articulo como su única base de datos de eventos y cada valor que se le agrega puede ser el cambio que se le hizo al articulo y al leer su historia se puede reconstruir a su última versión. Esto resulta de una práctica muy común en estas arquitecturas distribuidas.

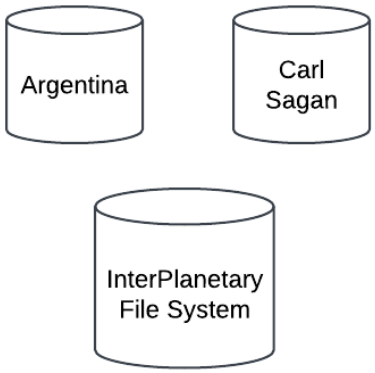

Figura 8: Representación de cada articulo como su propia base de datos.

De esta manera ya tenemos una representación para los artículos, pero faltaría una forma de saber que artículos existen actualmente. Es por eso que tenemos que agregar una última base de datos, también de eventos, que represente la wiki en si, con los nombres de los artículos existentes. De esta manera un usuario puede saber, al acceder a esta base de datos representando un repositorio de artículos, cuales son los artículos que existen y solo acceder a la base de datos correspondiente a ese articulo, sin necesidad de replicar la totalidad de los artículos existentes, algo que sería inviable.

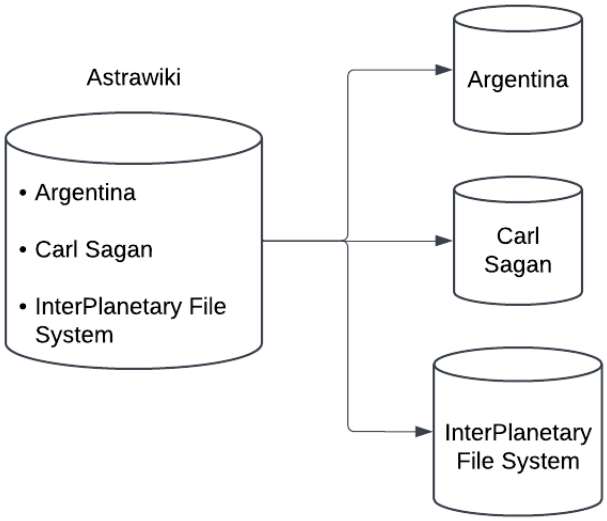

Figura 9: Representación del repositorio de artículos como base de datos.

Ahora bien, ¿cómo podemos acceder a la base de datos? En OrbitDB las bases de datos tienen una dirección que las identifica, formada en su creación. Esta dirección está conformada por el nombre de la base de datos, su tipo y su *Access Controller*. Este último sirve para indicar quien tiene acceso y permisos sobre la base de datos. Como en nuestro caso estamos permitiendo que todos tengan permiso, el *controller* va a ser siempre el mismo. Esto nos es importante, ya que, como el tipo de la base de datos y su *controller* van a ser siempre los mismos, con solo saber el nombre identificador, al crear la base de datos, vamos a tener siempre la misma dirección, y en consecuencia, vamos a tener siempre la misma base de datos.

```
/orbitdb/zdpuAmrcSRUhkQcnRQ6p4bphs7DJWGBkqczSGFYynX6moTcDL
```

Figura 10: Ejemplo de la *address* de una base de datos de OrbitDB.

Esto nos es de mucha utilidad ya que con saber únicamente el nombre identificador de la wiki, podemos acceder su base de datos, como también a las bases de datos de sus artículos al componer ambos nombres, el de la wiki con el del articulo, y asi acceder a la base de datos del artículo.

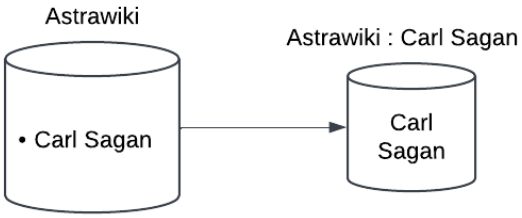

Figura 11: Representación de los nombres identificadores de la wiki y sus artículos.

Es importante notar que en ningún momento vamos a estar abriendo explícitamente una base de datos desde un nodo nuevo, sino que vamos a estar creando la misma base de datos vacia desde cero para luego sincronizarla con el resto de nodos. Esto sucede ya que, si recordamos, OrbitDB es

una base de datos **eventualmente consistente**, lo que significa que OrbitDB te asegura que en algún momento todos los nodos van a converger a la misma información, pero no te da la certeza de cuándo va a suceder. Esto también nos ayuda ya que no podemos nunca estar seguros si somos los primeros al crear una base de datos o si ya existe y aún no fuimos sincronizados, generándose como si fuera un *chicken egg problem*, por lo tanto al tener en cuenta se evita hacer asunciones que pueden generar problemas.

Otro punto importante es que al estar identificando a las bases de datos con su nombre, significa que es muy fácil crear una wiki alternativa que sea totalmente independiente a otras existentes, permitiendo asi total libertad de crear y mantener otras wikis que se quiera, cada una con sus articulos correspondientes y persistida por sus propios colaboradores. Esto nos va a ser de especial utilidad cuando este enfoque no solo lo utlicemos para wikis, el cual explicaremos con detalle más adelante.

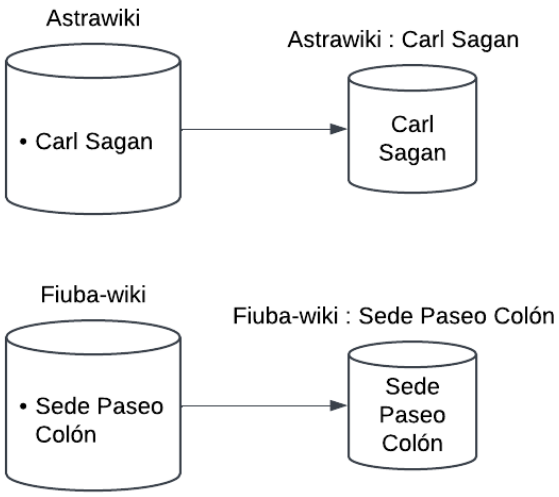

Figura 12: Representación de múltiples wikis independientes.

Colaboradores En orbitdb, al ser una base de datos peer-to-peer, distribuida y sin un servidor central, significa que la base de datos existe en cada peer que la componga, por lo tanto conectarse a una base de datos significa replicar la totalidad de su información de otros peers que nos la provean. De no estar esos peers, significaría que la base de datos no puede ser accedida y su información podría perderse, tal como sucedía cuando explicamos la infraestructura anterior. Es por eso que una gran parte de la arquitectura está pensada al rededor de peers que opten por ser colaboradores, estos peers van a replicar todas las bases de datos que existan actualmente en la base de datos central, pensandolo en el caso del repositorio, van a preservar todos los articulos que existan. De esta manera logramos que mientras haya por lo menos un colaborador en linea, la wiki va a poder ser accedida.

Idealmente no tendría que ser necesario replicar toda la información en cada colaborador, sino que replicarla inteligentemente en suficientes colaboradores proveeria de suficiente persistencia y disponibilidad. Sin embargo y similar a como se explicó anteriormente en el análisis de la infraestructura de despliegue, debido a la falta de *proof of storage* no podemos estar seguros nunca que una base de datos este siendo realmente replicada y por la tanto no podemos permitirnos dividir el almacenamiento utilizado entre distintos colaboradores, ya que seria muy susceptible a la perdida de información si se le asigna la responsabilidad de una base de datos a todos nodos los cuales no están verdaderamente replicandola. Esta decisión conlleva a otros puntos a remarcar de los cuales entraremos en detalle mas adelante.

Estos peers colaboradores son el punto más importante que diferencia esta solución con el resto

y que sigue con la filosofía de las aplicaciones comunitarias que estamos analizando, permitiendo que la disponibilidad de la información se este logrando a través de la donación de almacenamiento en vez de dinero.

Es importante remarcar que los colaboradores no tienen ningúna responsabilidad o permisos extra que los usuarios normales no tengan, su única función es ayudar con la disponibilidad de la base de datos, cualquiera que quiera puede colaborar.

Entonces, un usuario que quiera conectarse a la base de datos y obtener su información, por ejemplo querer conectarse a la wiki y ver artículos, debe primero conectarse a alguno de estos colaboradores, de los cuales puede replicar la información a su base de datos propia. Sin embargo esto no es tan trivial como parece, pasemos a ver como se resuelve este problema.

Manejo de conexión OrbitDB no se responsabiliza de manejar las conexiones, tampoco le importa, ya que te asegura que eventualmente la base de datos va a estar sincronizada entre peers, aunque se caigan las conexiones o te conectes mas tarde, todo se va a sincronizar sin conflictos eventualmente. Por lo tanto delega esa responsabilidad a **Helia**[34] la cual es la implementación de IPFS en los lenguajes javascript/typescript, que a su vez delega la responsabilidad a **LibP2P**[60] y de la cual tenemos que hacer uso nosotros para manejar las conexiones.

LibP2P es una colección de protocolos y utilidades para facilitar la implementación de una red peer-to-peer. Al crear un nodo de LibP2P se tiene que elegir como conformarlo en base a un conjunto modular de herramientas, dentro de los que se encuentran mecanismos de seguridad, de transporte, descubrimiento de pares, entre otros. Cada una de estas herramientas es importante y hace que el nodo funcione como queramos, más adelante explicaremos la decisión para cada herramienta elegida, sin embargo ahora nos vamos a centrar en dos protocolos de interés necesarios para solucionar el problema de manejo de conexión. Los protocolos de transporte y de descubrimiento de peers.

Los protocolos de transporte son los encargados de la comunicación entre nodos, de manera similar a la capa de transporte presente en toda red convencional. Se basan en tipos de transporte ya existentes, adaptados al uso peer-to-peer. Como lo son TCP, QUIC o WebSockets. Sin embargo no todos los transportes nos son útiles y la decisión de cual protocolo utilizar no es sencilla.

Uno de los requisitos de esta infraestructura era la posibilidad de utilizarla para crear aplicaciones que se puedan utilizar desde un entorno web, como es el caso de un usuario queriendo acceder a la wiki desde un navegador. Lo que sucede es que el entorno web no es muy amigable con las conexiones. Los navegadores están construidos sobre HTTP(S), un protocolo sin estado basado en solicitudes y respuestas. El cliente (navegador) envía una solicitud y luego espera una respuesta. Las conexiones se manejan en la capa de transporte y no mediante HTTP(S). Los navegadores imponen reglas estrictas, como los requisitos de certificados y el bloqueo de políticas de origen cruzado (cross-origin).

Es por esto que por razones de seguridad, no es posible que un navegador establezca una conexión TCP o QUIC sin procesar directamente desde el navegador, ya que todas las conexiones deben cumplir con los requisitos de Contexto Seguro (Secure Context), como por ejemplo que los mensajes se entreguen a través de TLS.

Es por esta razón que la infraestructura debe hacer una diferenciación entre nodos que se ejecuten desde un entorno independiente, como es el caso de **node.js**[66] y nodos que se ejecuten en un entorno web, únicamente para asignarle los protocolos de transporte correspondientes a su entorno.

Hay 2 conexiones que se tienen que resolver, la conexión entre nodos independientes y la conexión desde un nodo web a un nodo independiente.

Para los nodos independientes la solución es sencilla y el protocolo utilizado es TCP, con el cual pueden comunicarse entre sí y con otros nodos independientes de la red. El problema viene que, como explicamos, TCP no nos sirve para comunicar desde un nodo web a un nodo independiente, por lo tanto hay que explorar alternativas.

Al momento de comenzar con el desarrollo del repositorio de conocimiento, la implementación

de LibP2P en javascript ofrecía dos formas para resolver la conexión de web a nodo independiente.

La primera alternativa analizada y más conocida es la de utlizar **WebSockets**[91]. El protocolo WebSocket permite "secuestrar" una conexión HTTP y tras una solicitud para asegurar la conexión, el navegador obtiene acceso directo a la conexión TCP subyacente, permitiendo comunicación bidireccional persistente. Sin embargo esta alternativa no nos de fue de utilidad ya que para lograr esta mejora de conexión es necesario contar con un dominio o certificado por una autoridad certificadora, cosa que muy probablemente los nodos y usuarios utilizando la infraestructura no tengan.

La otra alternativa adicional considerada fue el uso de la técnica conocida como **Hole Punching**[35], una técnica muy interesante que permite que nodos privados —es decir, aquellos ubicados detrás de un firewall o en redes donde no pueden recibir conexiones entrantes, como hogares, redes corporativas o dispositivos móviles— puedan establecer conexiones con otros nodos. Para lograrlo, un nodo público actúa como intermediario, escuchando por ellos, este nodo es denominado como **Relay**. Esto nos sirve ya que un nodo web actua como un nodo privado ya que, aunque no pueden aceptar conexiones entrantes, sí pueden iniciarlas; así, todo el tráfico pasa a través del nodo intermediario, permitiendo una comunicación efectiva. No obstante, esta opción presenta una limitación relevante: el soporte para esta funcionalidad aún se encuentra en desarrollo en la implementación de libp2p para JavaScript, y no fue posible obtener un funcionamiento estable en el contexto de nuestro proyecto.

Al ver que ambas formas provistas no nos eran de utilidad nos puso en un lugar difícil, dado que de no poder lograr la conexión significaría que esta infraestructura no iba a poder funcionar, sin embargo el desarrollo e investigación continuó aunque no sabíamos si se iba a poder lograr una forma exitosa, lo cual era un riesgo. Sin embargo en al rededor de la mitad del proyecto, el soporte para una nueva alternativa se implementó.

Esta nueva alternativa es la conocida como **AutoTLS**[7] la cual conciste en obtener un certificado y dominio dependiente del ID del nodo usando el protocolo conocido como **ACME**[1], del cual libp2p provee un servicio DNS de bien público que responderá en nombre de cualquier usuario de Internet. De esta manera se puede usar el método de websockets de manera segura gracias al certificado que se obtiene. El funcionamiento de esta alternativa nos permitió continuar con el desarrollo y poder probar la infraestructura que se estaba creando. Sin embargo esta alternativa seguía teniendo algunas desventajas, la primera era que su conexión era muy inestable, al también ser una nueva alternativa experimental sufría de problemas. Pero más importante aún era que iba en contra de la filosofía que intentábamos lograr, que es la de una descentralización, como se requiere de un servidor de terceros para obtener el certificado, se depende de la disponibilidad de este servidor, y de no estar significaría que los nodos no puedan conectarse, por lo tanto aún necesitábamos una mejor alternativa.

Un tiempo después surge el soporte de una nueva alternativa, la cual ya teniamos conocimiento porque existía en otras implementaciones de libp2p en otros lenguajes y estaba en desarrollo para la implementación en javascript. Esta alternativa es la conocida como **WebRTC**[90] mas especificamente su implementación directa para conectar desde web a nodo independiente llamada **WebRTC-Direct**. Curiosamente hasta los mismos desarrolladores de libp2p recomendaban utilizar otras implementaciones en otros lenguajes para resolver la conexión web a nodo independiente hasta que el soporte de esta alternativa o otra muy similar se completen, citando: "Therefore, until WebRTC-Direct or WebTransport support is added to js-libp2p in Node.js, it's much easier to use go-libp2p."[21] Lo cual no podiamos permitirnos por el requisito de permitir utilizarlo en aplicaciones web y que orbitdb tambien se encuentra en el ecosistema de javascript.

WebRTC es un conjunto de estándares abiertos y APIs web que permiten a las aplicaciones web establecer conectividad directa para conferencias de audio y video, así como para intercambiar datos arbitrarios. LibP2P lo utiliza para lograr una conexión directa y sin necesidad de usar un certificado TLS. De esta manera logramos que ya no sea necesario utilizar AutoTLS y podemos conectarnos corrementamente desde un nodo web a un nodo independiente usando WebRTC-Direct.

De esta manera la topología de conexiones resultante entre nodos independientes y nodos web resulta de la siguiente forma:

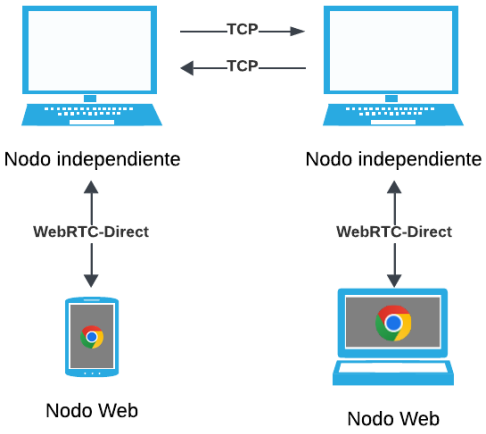

Figura 13: Topología de conexiones entre nodos independientes y nodos web.

Algo a notar es la falta de una conexión entre nodos web, lo cual sería algo muy útil, ya que los mismos usuarios web utilizando la aplicación en un determinado momento, podrian ayudar a proveer de las bases de datos a nuevos usuarios que se conecten y no dejar que solo los nodos independientes puedan hacerlo, lo cual seguría muy bien la filosofía de IPFS, sin embargo esto requeriría que los nodos web puedan escuchar conexiones entrantes y esto es únicamente posible haciendo uso de un **Relay** como explicamos anteriormente y que actualmente no fue posible obtener un funcionamiento estable en el contexto de nuestro proyecto. Igualmente sería algo muy fácil de implementar una vez que su funcionamiento sea estable y agregaría esta característica que mejoraría la infraestructura.

Descubrimiento de colaboradores Hasta ahora explicamos cómo se conectan y que protocolos de transporte utilizan los distintos nodos de nuestra infraestructura, sin embargo aún falta algo muy importante del manejo de conexión, el cual es cómo un nodo sabe a que nodo debe conectarse para poder sincronizarse y obtener la base de datos, en esencia, cómo descubre a los colaboradores a los que debe conectarse.

Una primera idea puede venir de dejar públicos algunas direcciones de nodos que sepamos que son colaboradores, por ejemplo nodos creados por nosotros, lo cual traería bastantes problemas. El problema más importante con este enfoque es que se estaría rompiendo completamente la descentralización de la infraestructura, ya que su disponibilidad pasaría a depender únicamente de la disponibilidad de estos *bootstrap nodes*, por lo tanto hay que lograr encontrar otro enfoque que respete con la descentralización que estamos buscando lograr.

Una alternativa más alineada con los principios de descentralización puede encontrarse al analizar cómo el sistema IPFS resuelve el problema de descubrimiento de contenido en su red. Como se explicó previamente, IPFS se basa en el concepto de content addressing [18], es decir, los datos son accedidos mediante su contenido (identificado de forma única por un Content Identifier o CID) en lugar de su ubicación específica. La localización del contenido se realiza a través de una Distributed Hash Table (DHT), donde los nodos que poseen determinado contenido se anuncian como proveedores de su correspondiente CID.

La DHT de IPFS funciona como una base de datos distribuida que permite localizar contenido de forma descentralizada. En lugar de un servidor central, cada nodo almacena una fracción del índice global que vincula CIDs con los nodos que los proveen. Así, cuando un nodo desea acceder a un determinado contenido, consulta la DHT para identificar los proveedores del CID correspondiente.[22]

Este mecanismo puede aprovecharse directamente para resolver el problema del descubrimiento de nodos colaboradores. Si consideramos que el CID representa la base de datos, construida a partir

de su nombre identificador, y los colaboradores son los proveedores de dicho CID, entonces basta con realizar una búsqueda en la DHT para obtener una lista de nodos actualmente activos que están ofreciendo ese contenido. De este modo, se logra un enfoque completamente descentralizado en el que los nodos colaboradores pueden anunciar su disponibilidad, y los nuevos nodos pueden descubrirlos e iniciar la sincronización sin depender de un punto central ni de direcciones de nodos preconfiguradas.

AstraDB Gracias al análisis anterior, se logró construir una implementación para el repositorio de conocimiento que mantuviera un estado y permitiera su modificación de una manera distribuida y descentralizada. Esta implementación era parte de la solución del repositorio de conocimiento y formaba una gran parte de su arquitectura, sin embargo al empezar con el desarrollo del mensajero en tiempo real, nos dimos cuenta que se podía crear una abstracción para que ambos casos de uso la utilicen, ya que todos los principios iban a ser los mismos, tanto la representación de los datos, como el manejo de conexión y uso de colaboradores. Es así como surge AstraDB.

AstraDB es la abstracción resultante de pensar la representación de los datos como pares clave valor. Donde en vez de tener un repositorio de artículos general, en donde se encuentre el nombre de todos los artículos y cada artículo como su propia base de datos, tengamos un base de datos general en donde se encuentren todas las claves que existen actualmente y una base de datos por cada clave con el contenido de dicha clave. Si bien sería similar a como se compone una base de datos clave valor, esta tiene una importante diferencia, la cual es que los valores de clave se agregan a los anteriores en vez de remplazar el valor actual, esto sucede ya que se sigue respetando el uso de *append only* que nos permitía no tener que generar permisos para evitar el borrado de información y lograr una descentralización.

Su interfaz consiste de dos métodos principales, los cuales permiten el agregado de una nueva clave y obtención de los valores asociados a esa clave. Permitiendo de esta manera que la creación de nuevos casos de uso orientados al uso colaborativo y siguiendo los principios mostrados sea mucho mas fácil y rápido, logrando que el foco del desarrollo este en los requisitos del mismo y no en el funcionamiento del ecosistema.

Lo que se tiene que hacer es adaptar el caso de uso al paradigma de clave valor. En el caso del repositorio de conocimiento, como se explicó anteriormente, cada clave representa un artículo, y el contenido del mismo es cada actualización que se le realizó al mismo. Para el caso del mensajero en tiempo real, cada clave representa un determinado chat, con el nombre del chat como la clave, y su contenido representa cada mensaje enviado secuencialmente. Mientras el caso de uso se pueda representar con dicho paradigma, se puede utilizar astradb para su persistencia de forma colaborativa.

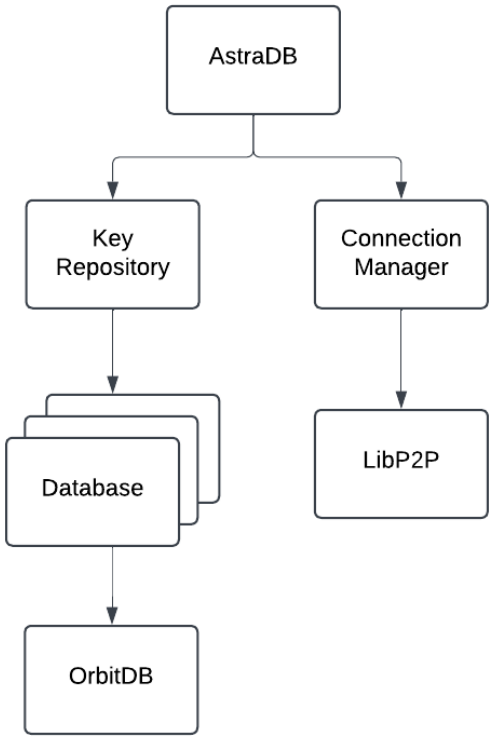

Figura 14: Arquitectura de AstraDB.

Su arquitectura, al igual que la descrita anteriormente, consiste de 2 principales ramas, una encargada del manejo de datos y otra del manejo de conexión. Al iniciar el nodo, este inicializa tanto el *Key Repository*, como el *Connection Manager* los cuales funcionan independientemente uno del otro y en paralelo para lograr en conjunto, el funcionamiento de AstraDB.

Key Repository Dentro del *Key Repository* se encuentra todo el manejo de datos de la infraestructura. Se encarga de mantener un registro de todas las claves existentes y de realizar las modificaciones requeridas.

Al inicializarse crea la base de datos central con el nombre identificador recibido por parámetro e intenta sincronizarla con los colaboradores conectados. Si no logra sincronizarse con ningún colaborador se asume que es una nueva base de datos y continua con su funcionamiento.

Si al crear el nodo de AstraDB se optó por ser colaborador, al recibir una actualización de la base de datos central sobre una nueva *key* creada, la base de datos representando el contenido de la *key* será abierta y guardada. De esta manera se logra la persistencia de la *key* y perimite que usuarios puedan acceder a ella al sincronizarse con este nodo.

Las base de datos utilizadas por el *Key Repository*, tanto la central como las de cada *key* individual, están representadas por una abstracción llamada *Database*. En esta abstracción se maneja toda interacción con la base de datos de OrbitDB.

Al incializarse se debe optar por si se desea sincronizar la base de datos o si no es necesario. Si se sincroniza entonces se va a esperar que la base de datos se sincronice con algun colaborador conectado, lo que significa obtener todas las actualizaciones que contiene la instancia de base de datos contenida en el colaborador. Una vez sincronizada esta ya permite su funcionamiento, el agregado de nuevo contenido a la base de datos u obtención de todo el contenido del mismo.

OrbitDB permite escuchar por eventos emitidos por la instacia de la base de datos, en nuestro caso es de suma importancia cuando se recibe una nueva actualización, que sucede al sincronizarse

con otro usuario y se recibe al menos un nueva entrada a la base de datos, sin embargo una actualización no es siempre el ultimo valor agregado a una base de datos. Por la naturaleza de las bases de datos de OrbitDB y su eventual consistencia, significa que al sincronizarse ambas bases de datos van a tener el mismo contenido y en el mismo orden. Aunque estemos usando una base de datos de tipo eventos y solo se pueda agregar al final, no significa que si dos de esas instancias de bases de datos tienen su contenido propio y aun no fueron sincronizados entre si, al sincronizarse puede que nuevo contenido quede por debajo de contenido nuevo. Es por esto que desde la abstracción se debe tomar recaudos para que no quede contenido nuevo sin notificar, independientemente de su vejez.

Esto es de suma importancia para cuando se utiliza para el mensajero en tiempo real, ya que AstraDB permite escuchar por actualizaciones en tiempo real para una determinada clave, recibiendo de esta forma todas las actualizaciones de un determinado chat.

Para lograr esto la abstracción *Database* mantiene un registro de todas las claves vistas hasta un determinado momento, en OrbitDB cada entrada a una base de datos tiene un hash identificador y es eso lo que nos guardarnos. Al recibir un evento de *Update* desde la base de datos de OrbitDB, debemos iterar todas las entradas que contiene dicha base de datos para encontrar si hay entradas nuevas, no antes vistas. Se debe iterar en su totalidad ya que nada asegura que no haya entradas nuevas en distintas posiciones de la base de datos, como vemos que puede suceder en el siguiente ejemplo:

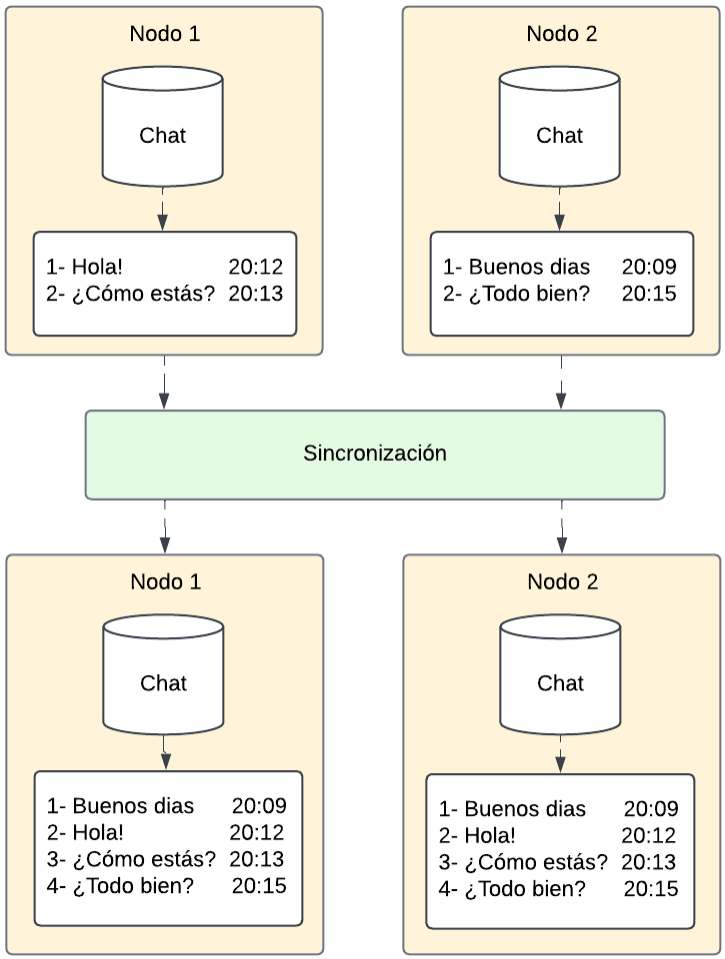

Figura 15: Ejemplo de sincronización de nodos.

En el ejemplo contamos con dos nodos independientes que forman parte de la misma base de datos con nombre 'Chat'. Cada nodo tiene su propia representación de la base de datos hasta un determinado momento. A continuacion se puede observar que ocurre una sincronizacion entre ambos nodos, esto hace que ambos nodos pasen a tener la misma representacion de informacion de la base de datos fusionando lo que tenian previamente, de esto se encarga OrbitDB. Lo importante a notar aca es que desde el punto de vista del nodo 1, las entradas nuevas de la base de datos no estan al final de la misma unicamente, sino que aparece una nueva entrada previa a sus entradas que tenia. Y como se mencionó anteriormente, OrbitDB solo anuncia el evento *Update* al sincronizarse con otro nodo y con únicamente la última entrada actualizada, en este caso avisaría del último mensaje con contenido '¿Todo bien?'. Es por esto mismo que desde la abstracción *Database* es necesario recorrer todas las entradas actuales para encontrar aquellas nuevas en la instancia de base datos propia y avisar por un evento individualmente por cada una.

Connection Manager Dentro del *Connection Manager* se encuentra todo el manejo de conexiones de la infraestructura. Se encarga de encontrar y conectarse a otros colaboradores que provean la base de datos utilizada, como también agregarse como proveedor si se trata de un nodo colaborativo.

Al inicializarse lo primero que hace es construir el **CID** identificador de la base de datos central, para esto utiliza **Helia**[34], la implementación de IPFS para javascript, para codificar el nombre identificador recibido por parámetro. Como se explicó anteriormente, este CID va a ser siempre el mismo para todas las instancias de AstraDB que utilicen el mismo nombre identificador.

Luego inicializa tres servicios que van a estar funcionando en paralelo durante la ejecución de AstraDB. Esos servicios son: *SearchForProviders*, *ProvideDB* y *ReconnectToProviders*.

En *SearchForProviders* se va a realizar una busqueda continua de nuevos proveedores de la base de datos que se está utilizando, para conectarse a ellos y poder sincronizarse. Para esto va a realizar una consulta a la **DHT** utlizando **LibP2P**[60], buscando los proveedores del CID construido anteriormente. Al encontrar un nuevo proveedor, se va a intentar conectarse a este.

En *ProvideDB*, solo en caso de optar por ser colaborador, se va a notificar utilizando la **DHT** que el nodo es un proveedor de la base de datos utilizada. De esta manera otros nodos pueden encontrar al nodo utilizado en su busqueda de proveedores.

Por último, en *ReconnectToProviders* se va a estar iterando una lista de proveedores previamente conectados para intentar reconectarse a ellos, por si ocurrió una falla en la conexión.

Resultado Como resultado del análisis y creación de esta infraestructura, se logró proveer de una herramienta para facilitar la creación de nuevas aplicaciones con enfoque comunitario, distribuido y descentralizado utilizando el ecosistema de IPFS. De la cual nuestros casos de uso Astrawiki y Astrachat la utilizan para su funcionamiento.

Limitaciones Sin embargo este enfoque presenta algunas limitaciones o aspectos a mejorar que son importantes destacar.

Para la solución implementada, la seguridad no fue una prioridad. En consecuencia se presentan algunos problemas que están relacionados con esto. La solución da total libertad a la creación y agregado de información a las bases de datos sin ninguna limitación, junto a que los colaboradores deben almacenar la totalidad de la base de datos, por lo que se explicó anteriormente de la falta de *Proof of storage*, lo cual de por si es un limitación. En consecuencia es muy fácil realizar un ataque de spam, usando el espacio almacenado de los colaboradores, al crear nuevas *Keys* o agregar contenido sin importancia a cada una de ellas. Esto podría mitigarse si se utilizara un consenso a la hora de aceptar un cambio o no a la base de datos, dado que si el cambio no se acepta, entonces no se distribuye y no se efectúa, sin embargo, como se explicó, esto requiere un foco de seguridad y para mostrar su uso en el ecosistema no fue el punto principal.

Relacionado con lo anterior, permitir la modificación libre de toda la información puede que no sea lo deseado para otro tipo de aplicaciones. Si se quisiera crear un blog personal o de microblog-

ging, se desearia que solo pueda moficar la información el dueño del blog y que los colaboradores solo ayuden con la persistencia del mismo. Esto puede ser una posible mejora y es posible si se puediera optar que las bases de datos de las keys tengan el permiso de moficiacion unicamente al usuario que las creó, como tambien referenciar a la *Address* de la base de datos creada, ya que en este caso si depede de su creador, por lo que se explicó en anterioridad en la representación de los datos.

Tambien tiene una limitacion con que no se pueden crear cosas privadas, dado que el almacenamiento esta en la red de ipfs y cualquiera puede acceder a el y la falta de encirptacion, hace que para casos de uso donde se requiera privacidad de informacion no es lo ideal.

8.3.3. Colaboración

Se creó una herramienta para facilitar la colaboración a nuestra wiki. El nodo es capaz de fijar los contenidos de Astraweb, nuestro front-end, y también mantener todas las bases de datos utilizadas por Astrawiki y Astrachat en la capa de aplicación. Para lograr esto, se diseñó un sistema con cuatro contenedores, cada uno asignado a uno de los tres casos de uso.

Para el pinning del sitio web en si, se utilizó el contenedor de IPFS Cluster con la instrucción de colaborar con el clúster dado por parámetro. Este parámetro es la dirección IPFS del archivo `service.json` que identifica a un clúster. Además, IPFS Cluster requiere de un contenedor de Kubo para manejar el manejo de archivos en la red de IPFS.

Para Astrawiki, se hizo uso del contenedor ofrecido por Astrawiki CLI, front-end que se verá más adelante. Con este contenedor, se puede actuar como colaborador fácilmente sin necesidad de interactuar con él.

Por último, Astrachat se levanta con un simple proyecto de Node que crea un nodo colaborativo y lo ejecuta en segundo plano.

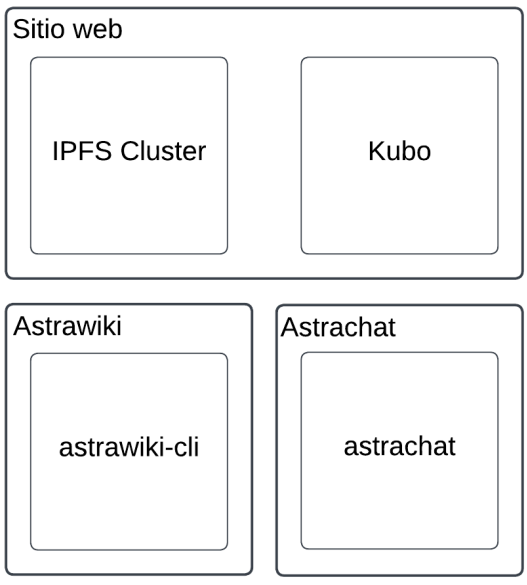

Figura 16: Arquitectura general del nodo colaborador

Cada uno de estos casos se puede seleccionar o deseleccionar desde un archivo de configuración `.env`, para que el usuario pueda elegir en que parte del proyecto contribuir. El resultado es una herramienta que, si bien fue diseñada para Astraweb, puede utilizarse para cualquier tipo de front-end que utilice Astrawiki y/o Astrachat.

8.4. Blockchain

Para el ecosistema de blockchain se decidió utilizar la red de Ethereum [94], al ser una blockchain popular y muy utilizada para el desarrollo de aplicaciones descentralizadas nos permite demostrar y comparar los casos de uso contra nuestra solución en IPFS. Ethereum está compuesta de nodos distribuidos que comparten poder de cómputo lo cual permite el desarrollo de aplicaciones descentralizadas. Cuenta con una moneda que funciona a modo de incentivo, es decir, que los nodos reciben ganancias por formar parte de la red. Esto conlleva a que los usuarios de la red necesiten pagar para utilizarla a través de transacciones.

8.4.1. Swarm

Existen varias soluciones de almacenamiento distribuido dentro del ecosistema blockchain, en este caso, para el desarrollo del sitio web estático optamos por Swarm [88] que corre sobre una *sidechain* de Ethereum. Swarm surgió como uno de los tres pilares de Ethereum para una web descentralizada [70]. Funciona por *content addressing* como IPFS e incluye un modelo de incentivos utilizando su propia moneda llamada BZZ.

Feed Los *feeds* en Swarm funcionan de manera similar a los IPNS de IPFS. Es un puntero, con CID fijo, a un archivo. Esto permite actualizar el archivo al que apunta un *feed* manteniendo un punto de entrada fijo al sitio web.

8.4.2. Ethereum

Para los casos de uso del repositorio de conocimiento y el mensajero en tiempo real necesitamos una herramienta que funcione de manera *read-write* y como Swarm solamente se encarga de archivos estáticos buscamos alguna alternativa dentro del ecosistema blockchain. Para esto terminamos usando Ethereum.

Figura 17: *Smart contracts* que intervienen en el repositorio de conocimiento

Ambos casos de uso resultaron muy similares en su resolución, haciendo uso del patrón de diseño *Factory*. Existe un smart contract Factory que crea otros smart contracts (Artículo o Chat, según el caso de uso).

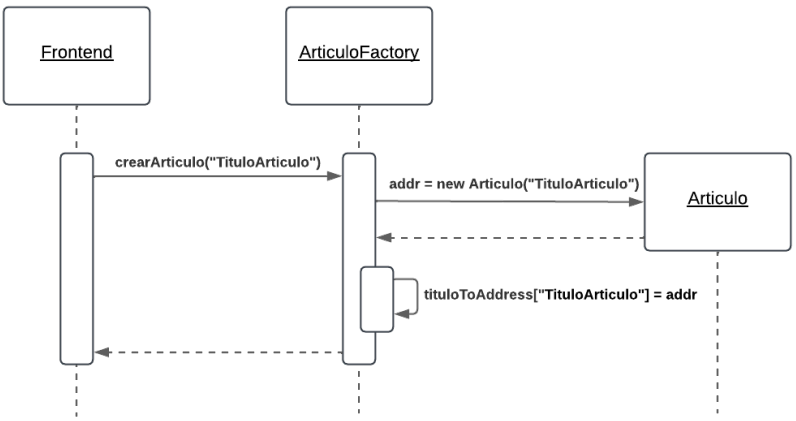

Figura 18: Creación de un artículo

De esta manera el *Factory* tiene un *mapping* con todos los artículos creados y las direcciones correspondientes para accederlos. Si se quisiera acceder a un Artículo en particular primero se tiene que consultar al *Factory* para obtener la dirección del mismo y, como cada artículo es un *smart contract* en sí mismo, se puede consultar o modificar su contenido directamente interactuando con el Artículo en particular como se puede ver en la Figura 19.

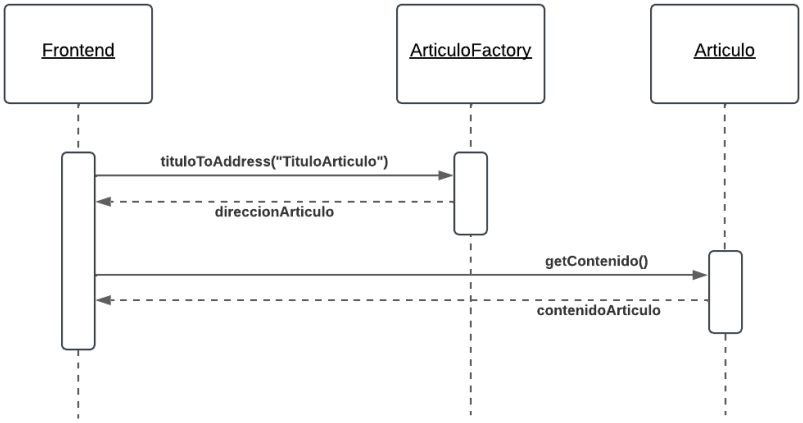

Figura 19: Obtención del contenido de un artículo

La principal diferencia entre el repositorio de conocimiento y el mensajero en tiempo real está en que los mensajes del mensajero tienen que ser vistos por los demás usuarios que participan de la conversación en el momento que se envían. Esto no es estrictamente necesario en el repositorio de conocimiento pero sí lo es en el mensajero.

Para afrontar este requisito se utilizaron los eventos de Solidity (el lenguaje de programación en el que se desarrollan los *smart contract* de Ethereum). Funciona de la siguiente manera, al momento de enviar un mensaje se emite un evento. Este evento se recibe en un listener que fue previamente inicializado al instante previo de haber obtenido el Chat en el frontend. Al recibir este evento el frontend puede actualizar la pantalla mostrando el mensaje nuevo sin necesidad de obtener todos los mensajes.

Por otro lado, para el mensajero en tiempo real necesitamos una manera de identificar a cada usuario. Para esto se hizo uso de las *wallets*. Cada usuario se identifica utilizando su *wallet*, que tiene una clave pública, que pasa a ser el identificador del usuario, y una clave privada la cuál es

necesaria para firmar transacciones en nombre del usuario, que en nuestro caso funciona a modo de contraseña. Además, para que la lectura de las conversaciones sean más usables, se agregó la posibilidad de generar un nombre de usuario asociado al identificador del mismo. A este nombre de usuario lo llamamos alias y es único para todos los Chats asociados a un mismo ChatFactory. Una vez el usuario se conecta con su *wallet*, puede elegir un alias y cambiarlo cuando desee siempre y cuando no exista actualmente algún otro usuario con ese mismo alias.

Finalmente, nos queda la funcionalidad de que un usuario pueda responder a otro mensaje. Primero necesitamos una manera de identificar a cada mensaje de manera unívoca. El identificador de cada mensaje se genera hasheando el timestamp del bloque, el identificador del emisor y la cantidad de mensajes en el chat en el momento que se envía. Luego, cuando se responde a otro mensaje se almacena el identificador del mensaje al que se está respondiendo (el mensaje padre) dentro de la estructura del mensaje que se está enviando. Todo esto se resuelve dentro del *smart contract* del Chat correspondiente.

8.5. Hyphanet

Hyphanet [14][42] es una plataforma peer-to-peer para publicar y comunicar, resistente a la censura y respetuosa de la privacidad.

Originalmente conocido como Freenet y creado como un trabajo profesional de fin de carrera por Ian Clarke, Hyphanet es una plataforma de software libre que permite compartir archivos, navegar y publicar sitios de forma anónima. Es descentralizado para hacerlo menos vulnerable a ataques, y de ser usado sólo con personas de confianza lo hace difícil de detectar.

Aplicaciones comunitarias La plataforma se basa en conexiones por nodos que gestionan la información que hay en la red. Estos nodos pueden conectarse a una red abierta llamada *Opennet* en la cual las conexiones se harán con nodos de cualquier persona de cualquier parte del mundo. Por otro lado es posible configurar el nodo para que se conecte sólo con aquellas personas que uno conozca, creando así una red "privada" (o *darknet*) en la que sólo personas de confianza puedan conectarse.

El contenido que se publica en los nodos permanece de forma encriptada y repartido en varias partes por distintos nodos. Siempre que un archivo sea solicitado el mismo será cacheado en los distintos nodos que lo soliciten.

No es posible "borrar" un archivo como tampoco es posible guardarlo a voluntad. Si un archivo no es suficientemente solicitado eventualmente no se puede recuperar más. Los nodos tampoco pueden elegir qué contenido (partes de un archivo) guardar o no ya que los mismos están encriptados. Si un archivo es subido por un nodo que no está disponible en este momento el archivo no se pierde porque ya fue distribuido por los demás nodos que estaban conectados a él.

8.5.1. Plugins

Es posible crear aplicaciones de comunicación con los llamados *plugins*. Estos *plugins* deben ser hechos en el lenguaje Java (o al menos el `main` debe estarlo) por decisión de diseño (alegando que Java es "más seguro"). No están aislados del sistema *host* por lo que pueden acceder a toda la información que quieran. Se deben compilar y proveer el `.jar` correspondiente y cada usuario que quiera utilizarlo debe instalarlo en su respectivo nodo.

Entre los plugins más usados en el ecosistema podemos nombrar:

WebOfTrust [47] Se autodenomina un spam filter pudiendo puntuar (*trust values*) a cada usuario de forma que los que tienen puntaje muy bajo son catalogados como spammers y cualquier contenido que los involucre será filtrado.

Sone [46] Red social similar a Facebook en el que se pueden subir imágenes, comentar, conectarse con otras personas. Usa WebOfTrust para identificar a cada usuario.

Freemail [43] Un servicio de email dentro de Hyphanet que también depende de WebOfTrust.

Freetalk [44] Sistema de foro.

No hay una receta para implementar estos plugins ya que la guía que existe está incompleta y hace años que no se actualiza (lo mismo pasa con los plugins en sí, llevan años sin actualizarse).

La documentación brilla por su ausencia y cada plugin hace uso de la librería de *Freenet* de una forma distinta lo cual hace difícil saber cuál es la forma correcta (de haberla) para crear un plugin desde cero.

Toda información que deba guardarse se debe hacer en una base de datos local administrada por el plugin (ya sea usando un archivo o una librería como podría ser *sqlite*[85]) ya que no existe una base de datos distribuida en el ecosistema que lo facilite. Esto hace que cada nodo tenga la información sólo de aquellos nodos con los que interactúa, mientras más lo haga más datos va a tener que persistir. Además es necesario que el plugin tenga forma de garantizar la integridad de los datos sino podrían ser fácilmente manipulados por algún nodo generando inconsistencias para la aplicación (que a su vez debería poder manejar).

8.5.2. Sitio web estático

Hyphanet posee un software propio para poder agregar sitios llamado *jSite*[45]. Con el mismo, basta con seguir las instrucciones en la documentación oficial. Una vez finalizada la creación del sitio, el programa devolverá un *hash* el cual será necesario para poder acceder al sitio.

Caso de uso Se logró levantar un sitio web estático que sólo contenía un HTML con un `"Hello World"` utilizando *jSite*[45] en el ecosistema.

8.5.3. Otros casos de uso

Debido a la escasa documentación y falta de estándares para desarrollar aplicaciones en el ecosistema, no se prosiguió con los demás casos de uso y se optó por investigar el ecosistema *Freenet*.

8.6. Freenet

Freenet[31] es una red peer-to-peer para servicios descentralizados, sin censura, en donde los usuarios tengan el control del contenido.

Creado por Ian Clarke, el mismo creador de *Hyphanet*[42], es una plataforma nueva que busca ser una computadora descentralizada en reemplazo de servidores centralizados. Está hecha en Rust y utiliza WebAssembly para ejecutar las aplicaciones.

Siguiendo las bases de Hyphanet, y a diferencia de IPFS, Freenet busca ser una *computadora distribuida* donde cada peer es capaz de ejecutar código que contenga estado el cual tendrá eventual consistencia con los demás peers. Parecido a su antecesor, un contrato puede cachearse en varios peers si este es lo suficientemente popular y dejará de cachearse si deja de serlo.

8.6.1. Arquitectura

Key-value Freenet es un *global key-value store* que se basa en la idea del *small-world routing*[84] para la descentralización y escalabilidad. Las *keys* son código WebAssembly en dónde se especifican:

- Qué valores están permitidos en la *key*.
- Bajo qué circunstancias el valor puede ser modificado.
- Cómo se puede sincronizar el valor eficientemente entre los *peers* de la red.

Contracts La base de la comunicación de las aplicaciones distribuidas son los *contracts*. Estos *contracts* son código Rust compilado a WebAssembly donde una clase debe cumplir con la interfaz del contrato (`ContractInterface`). Este es encargado de mantener la consistencia del estado de la aplicación en los distintos *peers*. Está pensado para que la actualización sea eficiente de modo que los *contracts* se actualizan en base a las diferencias.

Contracts vs Smart Contracts Los *contracts* de Freenet poseen ciertas similitudes y diferencias con los *smart contracts* de Blockhain.

	Contracts (Freenet)	**Smart Contracts (Blockchain)**
¿Descentralizado?	Si	Si
¿Mantiene estado?	Si	Si
¿Puede ejecutar código arbitratiamente?	Si (limitación local)	Si (limitación global)
¿Se distribuye el estado entre distintas instancias?	Si	No
¿Pago?	No	Si

Cuadro 1: Comparativa entre Contracts de Freenet y Smart Contracts de Blockchain

Comunicación Al momento de probar el ecosistema, una aplicación podía comunicarse con un contrato a través de un *web socket*. De esta forma se busca mantener el estado actualizado en la aplicación local ante un eventual cambio hecho por otro *peer*.

8.6.2. Diferencias con Hyphanet

Funcionalidad Hyphanet es un disco duro descentralizado mientras que Freenet busca ser una computadora descentralizada.

Interacción en tiempo real Freenet permite a los usuarios subscribirse a los datos y ser notificado inmediatamente ante algún cambio del mismo. Esencial para aplicaciones de chat o interacción en tiempo real.

Lenguaje de programación Hyphanet fue desarrollado en Java. Freenet está implementado en Rust haciéndolo más eficiente y pudiéndose integrar mejor a los distintos sistemas operativos (Windows, Mac, Android, etc).

Transparencia Freenet está pensado como un posible reemplazo a World Wide Web.

Anonimato La versión anterior fue diseñada con foco en el anonimato. La nueva versión no ofrece esta opción *out of the box* sin embargo es posible crear un sistema por encima que provea cierta anonimidad.

8.7. Front-end

Como prueba de la versatilidad de los ecosistemas, y aprovechando la creación de paquetes, se desarrollaron diferentes front-ends para las aplicaciones realizadas.

Astraweb El principal front-end que contiene el repositorio de conocimiento y a su vez el mensajero en tiempo real. Además, posee la opción de escoger el ecosistema al que se desea conectar.

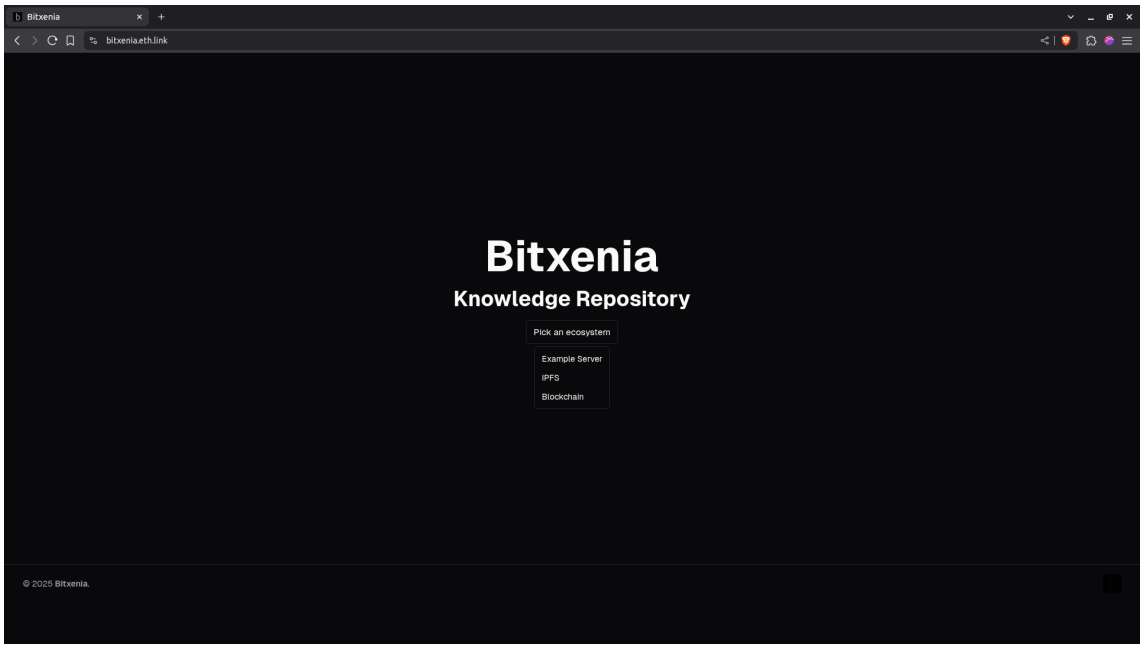

Figura 20: Página principal de Astraweb

Tecnología Se utilizó `React` [77] y `Next.js` [78] como *frameworks* para la creación de la aplicación web, basándonos en una plantilla llamada `rubix-documents` [82]. El código fue escrito en `Typescript`.

Servidor ejemplo Se desarrolló una solución centralizada como tercer ecosistema para este frontend, con dos propósitos:

1. Paralelizar el desarrollo del front-end y los distintos paquetes de cada ecosistema. Debido a que se definió una interfaz común tanto para el repositorio de conocimiento como para el mensajero en tiempo real, se logró avanzar con el front-end haciendo pruebas manuales con este servidor.
2. Comparar las implementaciones descentralizadas contra un enfoque centralizado.

El servidor fue creado en `Node.js` con `Express.js` como framework para interactuar con *requests* de HTTP.

Limitaciones Debido a la falta de un servidor tradicional para ofrecer el contenido, el uso de *Server Components* o componentes de servidor [83] no era posible. Esto implicó modificar ampliamente la plantilla utilizada para descartar este tipo de componentes en favor de aquellos que pueden ser compilados y luego utilizados por el cliente sin interacción con un servidor.

Astrawiki CLI Front-end de terminal, desarrollado para el caso de uso del repositorio de conocimiento y para el ecosistema de IPFS en específico. Cuenta con todas las funcionalidades del repositorio de conocimiento, como crear, editar y ver artículos, consultar versiones pasadas, e incluso colaborar, lo cual se verá en el apartado de IPFS. Además, cuenta con un contenedor `Docker` publicado, con el fin de fácilmente iniciar un nodo sin necesitar una instalación de `Node` y demás dependencias. Funciona como un `daemon` [19], es decir, se inicia y funciona en segundo plano hasta que se indique lo contrario.

Su propósito es, por un lado demostrar la versatilidad del modelo de paquetes utilizado para crear distintos front-ends. Y por otro lado, el contenedor es útil para el nodo colaborador desarrollado para IPFS visto previamente.

Arquitectura Se compone de un cliente, el cuál se inicia con cada comando (`start`, `add`, `get`, `edit`, `list`, etc.) y un servidor que se ejecuta por detrás, el cual inicia la instancia del repositorio de conocimiento de IPFS y utiliza su API cuando recibe `requests HTTP`.

Figura 21: Ejemplo de uso de `astrawiki-cli`

Astrachat CLI Frontend de terminal, desarrollado para el caso de uso de mensajero en tiempo real y para el ecosistema de Blockchain.

Figura 22: Página principal de `astrachat-cli`

Tecnología Se utilizó `React` [77] y `Ink` [48] como *frameworks* para crear la interfaz de este frontend.

Limitaciones Debido a que se ejecuta en la terminal no es posible hacer uso de una wallet como Metamask [63] ya que la misma sólo funciona en web. Debido a esto, se *hardcodearon* algunas de las wallets que provee Hardhat [33].

9. Metodología aplicada

La metodología aplicada para la gestión del proyecto fue una versión simplificada de Scrum. El desarrollo se dividió en sprints semanales para los cuales utilizamos un tablero Kanban en Github Projects donde se fueron agregando las tareas a realizar para cada caso de uso y ecosistema. Se realizaron reuniones semanales fijas que se usaron como punto de control, donde se revisó lo hecho durante la semana y definimos pasos a seguir para las siguientes. También nos fue útil para detectar posibles ajustes o cambios de rumbo que fueron surgiendo a lo largo del trabajo. Al finalizar cada una de estas reuniones realizamos minutas que nos sirvieron para resumir lo tratado en cada una y tener en claro los pasos a seguir después de las mismas.

Durante el descubrimiento de las funcionalidades de cada caso de uso realizamos *User Story Mappings* (USM) para organizar las tareas de realizar.

Los distintos artefactos que fueron surgiendo durante el desarrollo del trabajo fueron almacenados en un Google Drive compartido entre todo el equipo. Dentro del mismo se pueden encontrar: minutas de reuniones, cronogramas, USM, entre otros.

Para el control de versiones del código se creó una organización en Github en la que se fueron creando repositorios para los distintos paquetes que integraron el trabajo.

La modalidad fue virtual y asincrónica. Nos mantuvimos en constante comunicación a través de un servidor de Discord y también se realizaron sesiones de *pair* y *mob-programming* en distintas ocasiones.

10. Experimentación y/o validación

En base a los casos de uso implementados se realizó un análisis cualitativo y cuantitativo de cada ecosistema, con el objetivo de compararlos y ver las ventajas y desventajas de cada uno, teniendo en cuenta las siguientes categorías:

- **Costos**: ¿Cuánto nos cuesta desplegar y mantener un servicio en cada ecosistema? ¿Un usuario final necesita efectuar algún costo monetario para el uso de la aplicación?
- **Experiencia de desarrollo**: ¿Qué tan fácil es desplegar en cada ecosistema? ¿Existen herramientas y documentación necesario que facilite el desarrollo?
- **Viabilidad**: ¿Qué tan viable es crear una aplicación comunitaria para cada uno de estos ecosistemas?
- **Performance**: ¿Cuánto tiempo tarda la creación y obtención de elementos en cada aplicación?

10.1. IPFS

10.1.1. Costos

10.1.2. Experiencia de desarrollo

10.1.3. Viabilidad

10.1.4. Performance

La performance en IPFS se ve afectada por algunas variables. Entre ellas se encuentran:

- Uso de cache por parte de nodos de IPFS o gateways cuando se recupera un archivo.
- Cercanía al nodo correspondiente a la hora de publicar un CID en la Distributed Hash Table.
- Configuración y capacidades del nodo que tiene el contenido que se requiere.
- Cantidad de nodos alojando el contenido que se requiere.

Se puede minimizar el efecto de estas variables en la medida final sin distorsionar las métricas obtenidas. Más adelante se verán las maneras en las que se puede lidiar con estas variables.

Sitio Web Informativo La métrica que se decidió medir es la del **tiempo que tarda un nodo en desplegar un sitio web o contenido**. Para ello, se creó un clúster con un único nodo y un repositorio Git con contenido de distinta forma.

Obtención de las métricas En este caso, el proceso de despliegue se contiene dentro del contenedor `watcher`. Para medir el tiempo real que transcurre en cada paso del despliegue, se utilizó el comando de GNU `time` para cada paso, y el resultado es sumado para obtener el tiempo total que tardó desplegar el contenido.

Variables consideradas Las métricas obtenidas se lograron ajustando dos variables: el tamaño total del contenido, y la cantidad de archivos del mismo. Los archivos en sí fueron generados repitiendo un `UUID` hasta alcanzar el número de bytes deseados. Se utilizó este tipo de identificador para asegurar de que ningún archivo permanezca en alguna caché de la red o en la DHT, y a su vez no se repitan los CID entre archivos de la misma prueba.

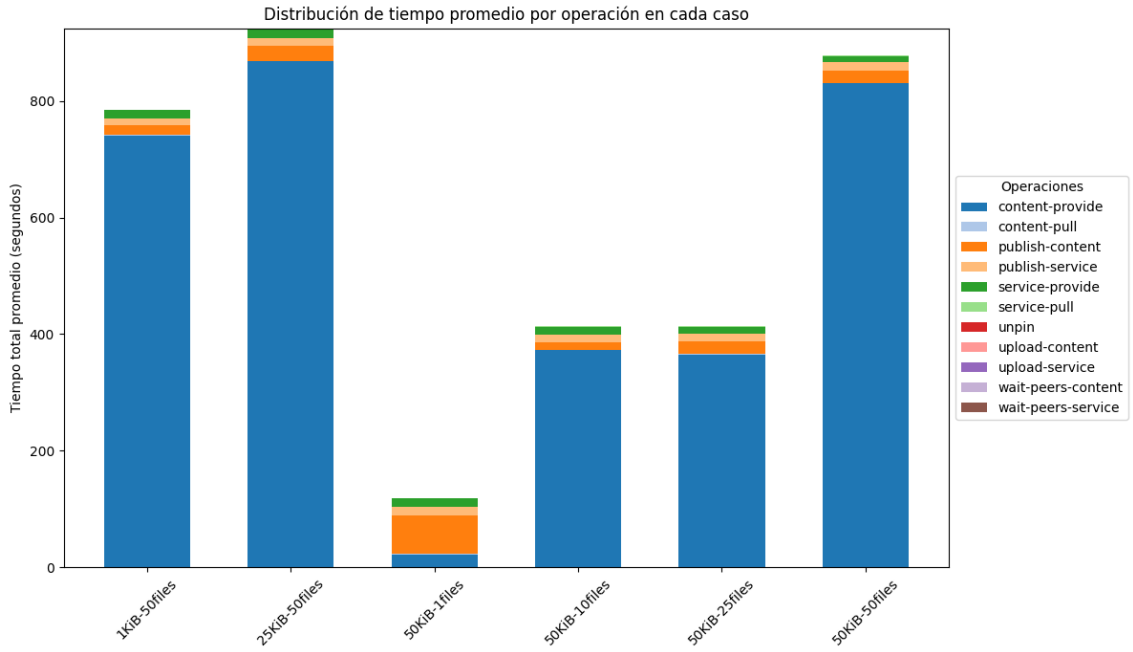

Figura 23: Distribución del tiempo promedio para desplegar el contenido para cada caso

De este gráfico podemos concluir que la operación más significante en términos de tiempo en el despliegue es la *providing* del contenido. Esto es, la publicación de todos los CIDs en la DHT. Luego, le siguen el providing del `service.json`, y las operaciones para actualizar el valor al que apunta el nombre de IPNS, tanto del contenido como del `service.json`. Las demás operaciones se pueden despreciar.

	Max	Mean	Min	Std	Median
1KiB-50files	23.65 s	17.66 s	7.70 s	6.39 s	21.71 s
25KiB-50files	67.29 s	25.92 s	8.96 s	15.24 s	22.95 s
50KiB-1files	78.75 s	66.09 s	61.48 s	5.83 s	62.93 s
50KiB-10files	30.57 s	12.93 s	7.10 s	7.42 s	8.96 s
50KiB-25files	23.79 s	22.65 s	21.94 s	0.46 s	22.57 s
50KiB-50files	23.36 s	20.95 s	8.17 s	4.27 s	22.43 s

Cuadro 2: Estadísticas para `publish-content`

	Max	Mean	Min	Std	Median
1KiB-50files	17.76 s	10.30 s	7.54 s	3.39 s	8.79 s
25KiB-50files	23.03 s	12.76 s	7.31 s	6.37 s	8.74 s
50KiB-1files	23.08 s	15.37 s	7.52 s	6.71 s	15.25 s
50KiB-10files	23.31 s	13.31 s	7.34 s	6.86 s	8.44 s
50KiB-25files	23.02 s	12.17 s	7.43 s	5.80 s	8.97 s
50KiB-50files	22.85 s	13.91 s	7.28 s	6.43 s	11.32 s

Cuadro 3: Estadísticas para `publish-service`

Teniendo en cuenta esto, se puede observar cuál es la causa de la variación del tiempo de providing ajustando las dos variables mencionadas.

	Max	Mean	Min	Std	Median
1KiB-50files	851.26 s	740.32 s	682.64 s	41.45 s	734.15 s
25KiB-50files	918.36 s	868.11 s	815.43 s	26.57 s	872.25 s
50KiB-1files	42.09 s	21.84 s	10.79 s	7.49 s	20.14 s
50KiB-10files	419.87 s	372.47 s	327.86 s	25.09 s	377.78 s
50KiB-25files	412.30 s	364.53 s	306.15 s	26.53 s	366.01 s
50KiB-50files	921.54 s	830.06 s	744.83 s	41.12 s	826.65 s

Cuadro 4: Estadísticas para `content-provide`

	Max	Mean	Min	Std	Median
1KiB-50files	27.62 s	14.83 s	8.84 s	6.12 s	11.41 s
25KiB-50files	34.34 s	15.67 s	10.99 s	6.64 s	11.66 s
50KiB-1files	25.71 s	13.63 s	6.49 s	5.26 s	11.34 s
50KiB-10files	20.88 s	12.86 s	10.84 s	3.11 s	11.30 s
50KiB-25files	20.78 s	12.42 s	9.78 s	3.47 s	11.26 s
50KiB-50files	20.27 s	11.31 s	6.82 s	3.17 s	10.97 s

Cuadro 5: Estadísticas para `service-provide`

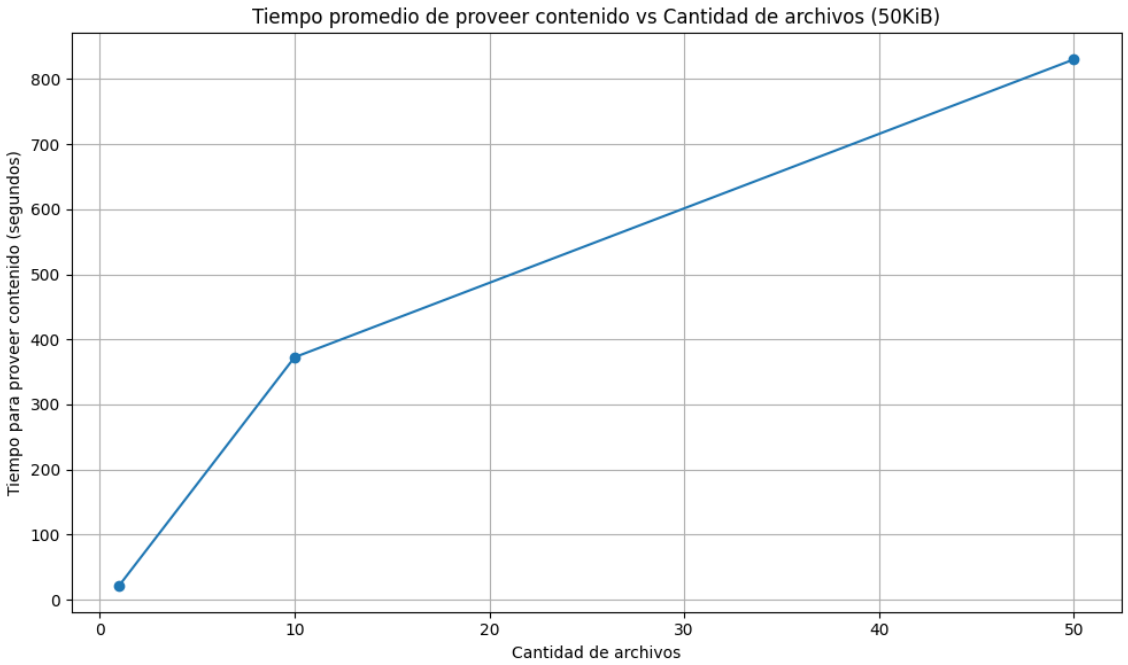

Figura 24: Tiempo promedio por cantidad de archivos

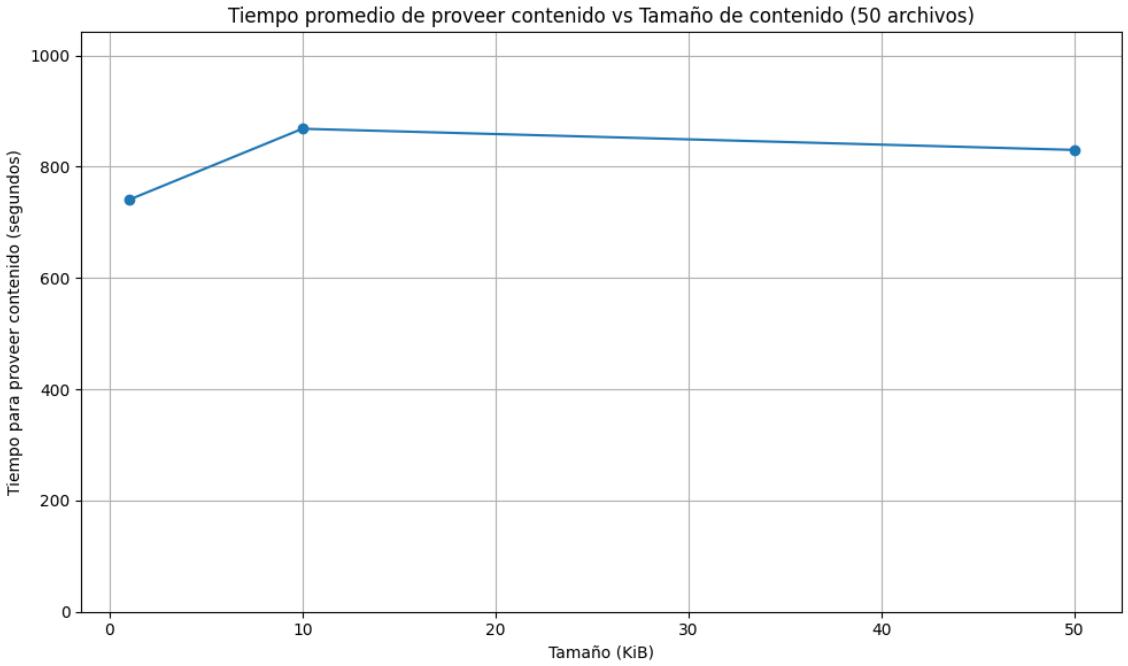

Figura 25: Tiempo promedio por tamaño del contenido

La cantidad de archivos que tiene el contenido a publicar es la variable que modifica drásticamente el tiempo que tarda un nodo confiable para desplegarlo, como se ve en las figuras. Esto se debe a que la publicación del CID del directorio no es suficiente para que otro nodo pueda obtener el contenido de ese directorio. En cambio, se requiere que se publique todos los archivos y directorios que componen el contenido. Esto también se demuestra con el tiempo constante para publicar el archivo `service.json`, ya que en todos los casos sigue siendo un sólo archivo.

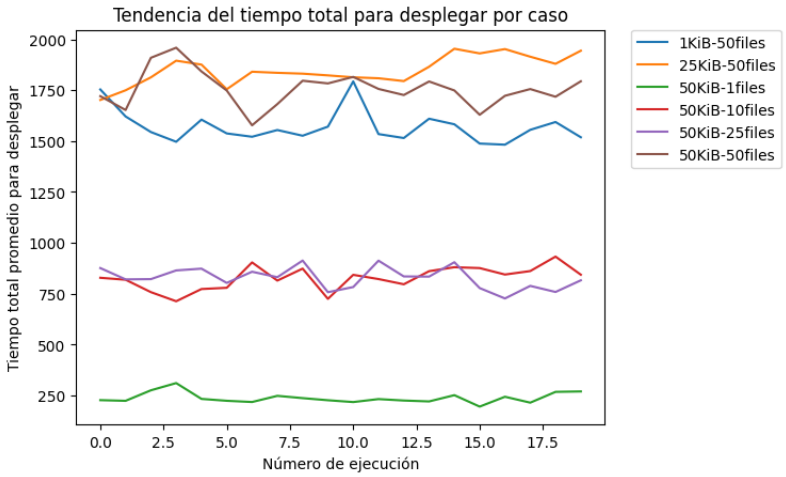

Figura 26: Tendencia del tiempo total promedio para desplegar el contenido en ejecuciones consecutivas.

Por último, se ve como la tendencia en ejecuciones consecutivas no afecta el tiempo que se tarda en desplegar.

Astrawiki Las métricas tanto de Astrawiki como Astrachat se ven muy relacionadas, debido a que el grueso del trabajo relacionado a IPFS se realiza con su biblioteca en común, AstraDB. Por

ejemplo, el tiempo que tarda un nodo en obtener una nueva versión de un articulo es el similar al que tarda un nodo en recibir un mensaje. Por ello, se prefirió complementar las métricas obtenidas para acentuar el uso específico de AstraDB de cada caso de uso, y evitar métricas repetidas.

Astrachat

10.2. Blockchain

10.2.1. Costos

Swarm Al deployar el sitio web es necesario contar con *postage stamps* que son la manera de pagar por el uso del almacenamiento en Swarm. Cada actualización que se realice al sitio requiere de *postage stamps* y, además, estos tienen fecha de vencimiento por lo que es necesario volver a pagar frecuentemente. Hay que tener en cuenta que dichos *postage stamps* se pagan en la criptomoneda BZZ que fluctúa de valor con respecto al dólar estadounidense. La obtención del sitio web no requiere de costo alguno, por lo que desde el punto de vista de un usuario lector de la aplicación no sería necesario pagar.

¡TODO: medir cuánto es el costo aproximado en USD o BZZ¿

Ethereum Se utiliza la moneda ETH para pagar por cada transacción, esto incluye tanto el despliegue de cada *smart contract* como también cada modificación al estado de los mismos. Por lo tanto, el usuario final de la aplicación termina pagando por la creación y edición de cada artículo en el repositorio de conocimiento, y por cada mensaje enviado en el mensajero en tiempo real. Por otro lado, para las operaciones de lectura no se tiene que pagar nada. A este monto que tiene que pagar el usuario se lo llama gas y varía con respecto a las operaciones que realiza el método que se ejecuta al llamar a un *smart contract* y, también, con respecto a la congestión de la red, es decir a mayor congestión mayor será el costo.

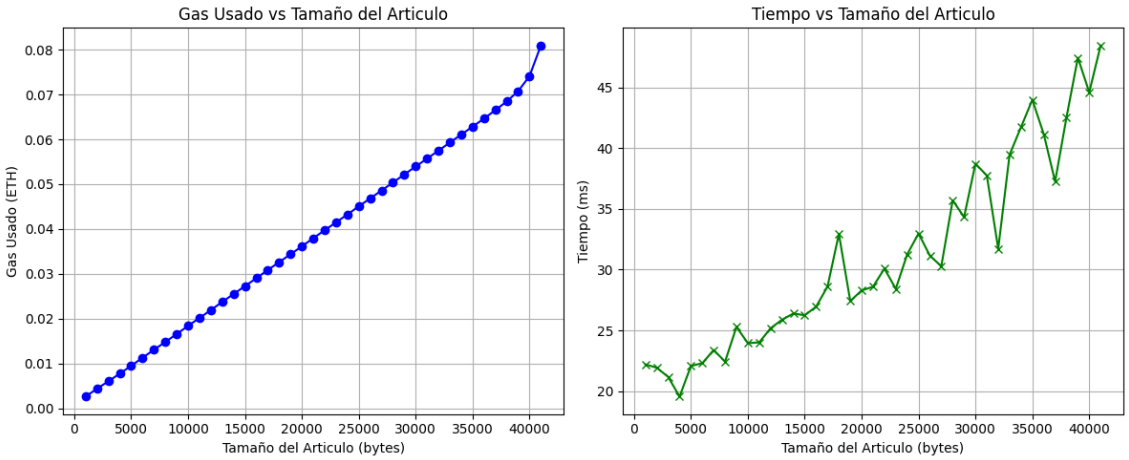

Figura 27: Tiempo y gas usado al crear artículos con tamaño de bytes creciente

10.2.2. Experiencia de desarrollo

Swarm En Swarm existe la herramienta de terminal swarm-cli [87] con la cual se puede interactuar con un nodo de Swarm. También el equipo de Swarm provee una Github Action que permite la posibilidad de automatizar el despliegue generando un pipeline que utilice dicha herramienta.

En cuanto a un ambiente de pruebas o *staging*, si bien no existe un *gateway* público que interactúe con la *testnet*, es posible levantar uno propio que sí lo haga apuntando a la *testnet* de Sepolia usando la herramienta gateway-proxy [32].

Ethereum Con la librería web3.js se puede interactuar con un nodo de Ethereum y realizar un despliegue de la aplicación. Además, con las herramientas de Hardhat se puede levantar una red de prueba que facilita el desarrollo local.

10.2.3. Viabilidad

Swarm Resulta más conveniente para sitios web o recursos estáticos, al igual que IPFS. Por otro lado, al ser una tecnología de almacenamiento no es posible la ejecución de código.

A diferencia de IPFS, Swarm cuenta con incentivos incluidos (por medio de la moneda BZZ), esto significa que para deployar contenido en la red es necesario pagar. Al hacerlo te asegura que el mismo va a estar disponible durante el tiempo equivalente al costo pagado, es decir, que no es necesario pinear los archivos puesto que se encuentran en la red con un TTL.

Ethereum Su punto fuerte es la ejecución de código, por lo cual es útil para funcionar como backend de aplicaciones web. Como hemos visto, por el costo de almacenamiento de los smart contracts, no es recomendable para recursos como imágenes, videos o incluso strings de texto muy largos como lo realizado en el repositorio de conocimiento.

Los eventos pueden resultar útil para la interacción en tiempo real requerida en el mensajero, pero lo positivo de esto queda opacado por el hecho de necesitar pagar por cada interacción, en el caso del mensajero por cada mensaje enviado. Esto se puede volver costoso rápidamente, además de tedioso al momento de utilizar la aplicación. Se pueden explorar alternativas para reducir esta fricción, como por ejemplo, que el contrato tenga un balance de tokens para ser gastados, lo cual haría que el usuario no tenga que confirmar cada transacción de mensaje enviado si no que directamente el contrato lo extrae de su balance; entre otras posibilidades.

10.2.4. Performance

Astrachat Las siguientes métricas se obtuvieron levantando una instancia local de Hardhat [33] y ejecutando 850 muestras, 3 veces cada una y con 3 mensajes de largo distinto:

- *short*: mensaje de 1 palabra. "*Lorem*"
- *medium*: mensaje de 10 palabras. "*Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam elementum.*"
- *large*: mensaje de 30 palabras. "*Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam ex risus, porttitor sed lacus id, egestas lobortis purus. Curabitur consectetur metus ut est vehicula egestas. Class aptent taciti sociosqu ad.*"

El tamaño de las palabras elegidas representa el largo de los mensajes más comunes que suelen enviarse en un chat.

Hay que tener en cuenta que los tiempos obtenidos corresponden al que le toma a la transacción ejecutarse en una máquina local. En un caso real, existe un tiempo mayor de red y de iteracción del usuario con la *wallet* para confirmar la transacción.

A partir de los datos obtenidos se calcularon el máximo (**Max**), mínimo (**Min**), la media (**Mean**), el desvío estándar (**Std**) y la mediana (**Median**).

Tiempo en enviar un mensaje La primer métrica tomada corresponde al tiempo que tarda en enviarse un mensaje.

Se puede observar que los tiempos entre los tamaños de mensajes no varían demasiado.

	short	medium	large
Max	10.93 ms	10.34 ms	10.68 ms
Mean	6.12 ms	6.26 ms	6.44 ms
Min	5.09 ms	5.24 ms	5.36 ms
Std	0.74 ms	0.60 ms	0.70 ms
Median	6.01 ms	6.20 ms	6.37 ms

Cuadro 6: Tiempo en enviar un mensaje

Gas usado para enviar un mensaje La siguiente tabla muestra el valor (en ETH y USD) de enviar el mismo mensaje corto. El precio tomado para la conversión de ETH a dólar es el de la fecha del 5 de junio de 2025 a las 19:00hs de $2439.17 de la página Coinmarketcap.

	short		medium		large	
	ETH	**USD**	**ETH**	**USD**	**ETH**	**USD**
Max	0.00042043	1.03	0.00060668	1.48	0.00085774	2.09
Mean	0.00034257	0.84	0.00051325	1.25	0.00074335	1.81
Min	0.00034221	0.83	0.00051274	1.25	0.00074264	1.81
Std	0.00000331	0.01	0.00000434	0.01	0.00000579	0.01
Median	0.00034221	0.83	0.00051274	1.25	0.00074264	1.81

Cuadro 7: Precio y gas usado para enviar un mensaje

Según los resultados obtenidos se puede decir que para enviar un mensaje cuesta más de 1 dólar.

Tiempo en obtener mensajes Para esta métrica se tomó el tiempo en obtener todos los mensajes para un chat el cual iba teniendo cada vez más mensajes (desde 0 hasta 850 mensajes). El gráfico muestra para los 3 tipos de mensajes, cómo el tiempo (en milisegundos) se va incrementando a medida que hay más mensajes en el chat.

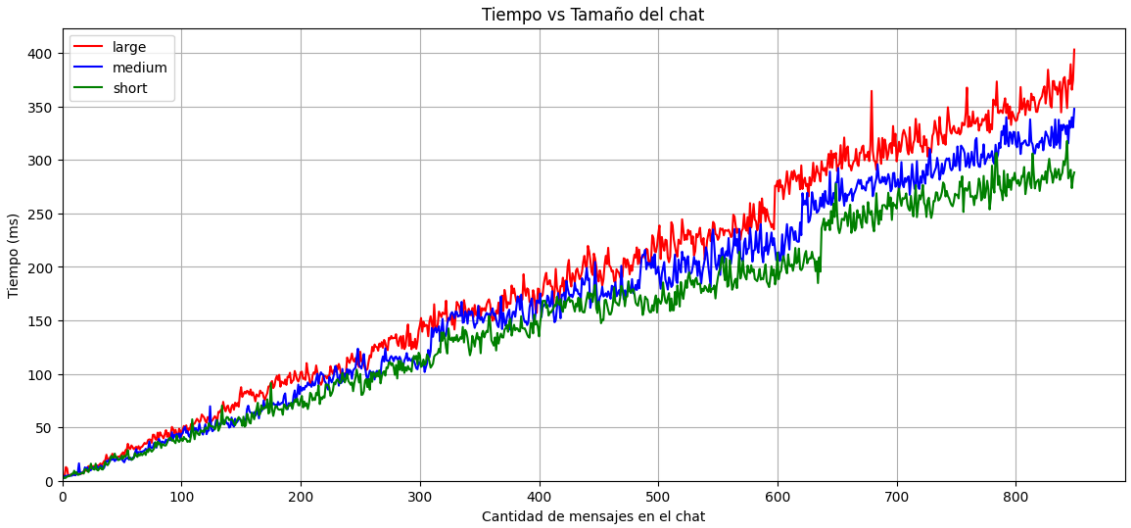

Figura 28: Tiempo para obtener mensajes de un chat según el tamaño del chat

Tiempo entre enviar y recibir un mensaje (mismo usuario) En esta métrica se midió el tiempo que tarda en un mensaje desde que es enviado a ser recibido por el canal de escucha de nuevos mensajes para un mismo usuario.

	short	medium	large
Max	38.12 ms	28.97 ms	24.66 ms
Mean	5.42 ms	5.76 ms	5.76 ms
Min	4.26 ms	4.31 ms	4.46 ms
Std	1.62 ms	1.74 ms	0.94 ms
Median	4.66 ms	5.05 ms	5.6 ms

Cuadro 8: Tiempo en enviar y recibir un mensaje (mismo usuario)

Tiempo entre enviar y recibir un mensaje (distintos usuarios) Esta métrica calcula el tiempo que tarda un segundo usuario en recibir un mensaje enviado por un primer usuario.

	short	medium	large
Max	13.19 ms	10.92 ms	13.95 ms
Mean	5.61 ms	5.84 ms	5.96 ms
Min	4.59 ms	4.53 ms	4.65 ms
Std	0.97 ms	1.28 ms	0.72 ms
Median	5.53 ms	5.06 ms	5.89 ms

Cuadro 9: Tiempo en enviar y recibir un mensaje (distintos usuarios)

Si comparamos las tablas de tiempos de enviar y recibir un mensaje (8 y 9) vemos que los tiempos son parecidos. Si bien los máximos de la primer tabla son mayores que los de la segunda y los datos están más dispersos, para los demás valores de la segunda tabla estos tienden a ser ligeramente mayores. En este caso particular, si se hiciera una prueba entre 2 usuarios de distintas partes del mundo el tiempo de conexión seguramente pesaría más.

10.3. Resumen

Resumiendo, y a modo de comparación entre ecosistemas, podemos concluir con lo siguiente:

	IPFS	**Blockchain**
Costos	Bajos o nulos	Escala con el uso de la aplicación
Desarrollo		Existen herramientas que facilitan el desarrollo y el despliegue
Viabilidad		Sitios estáticos y aplicaciones CRUD
Performance		

Cuadro 10: Comparación entre los ecosistemas de IPFS y Blockchain

11. Cronograma de las actividades realizadas

Inicialmente realizamos un cronograma tentativo en forma de diagrama de Gantt de la totalidad del trabajo, incluyendo el desarrollo de cada caso de uso, el despliegue en cada ecosistema y su documentación asociada como se muestra en la Figura 29. Sin embargo, a medida que fuimos desarrollando el trabajo nos encontramos con que dividir el trabajo por ecosistemas era más eficiente que dividirlo por casos de usos como en nuestro cronograma estimado.

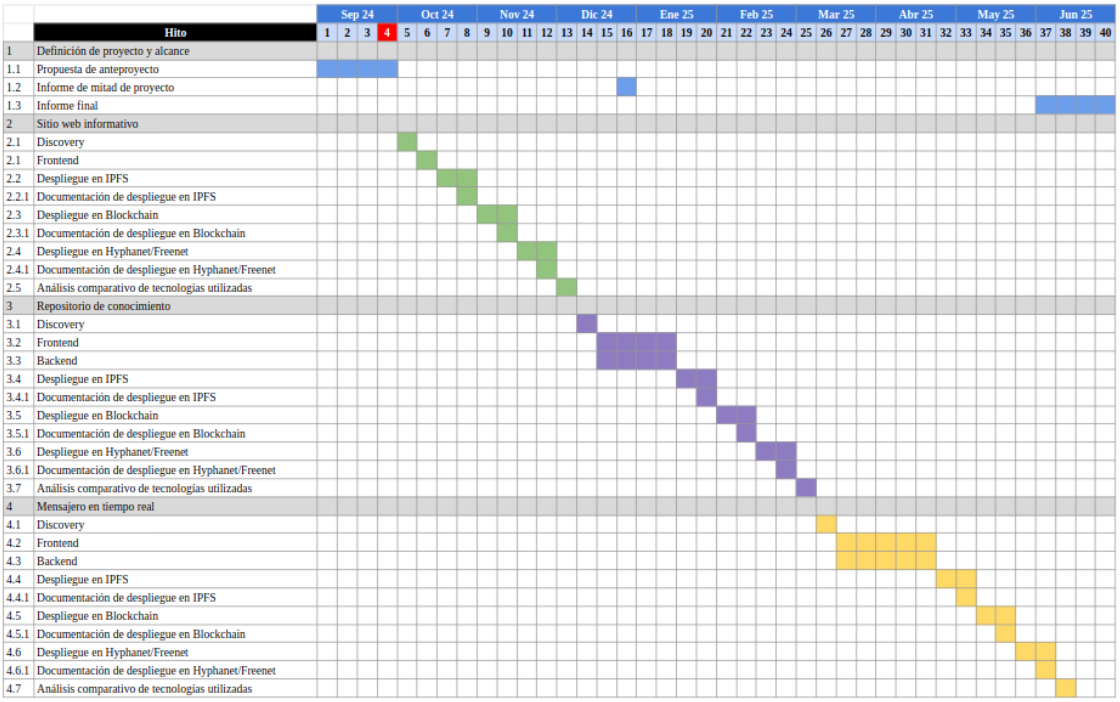

Figura 29: Cronograma tentativo

Por lo tanto, lo que terminó sucediendo es lo que se muestra en la Figura 30. La división se hizo por ecosistema, esto incluye la investigación, el desarrollo y la documentación de cada caso de uso para el ecosistema en cuestión. Hacerlo de esta manera nos permitió paralelizar los esfuerzos y evitamos superposiciones durante la investigación y desarrollo. Esta separación podría haber generado silos de conocimiento por ecosistema dentro del equipo, es por esto que las reuniones semanales de puesta en común, junto con la constante comunicación por chat y sesiones de *pair-programming*, fueron de vital importancia para mitigar dicha separación y que todo el equipo esté al tanto de los conocimientos adquiridos de cada ecosistema.

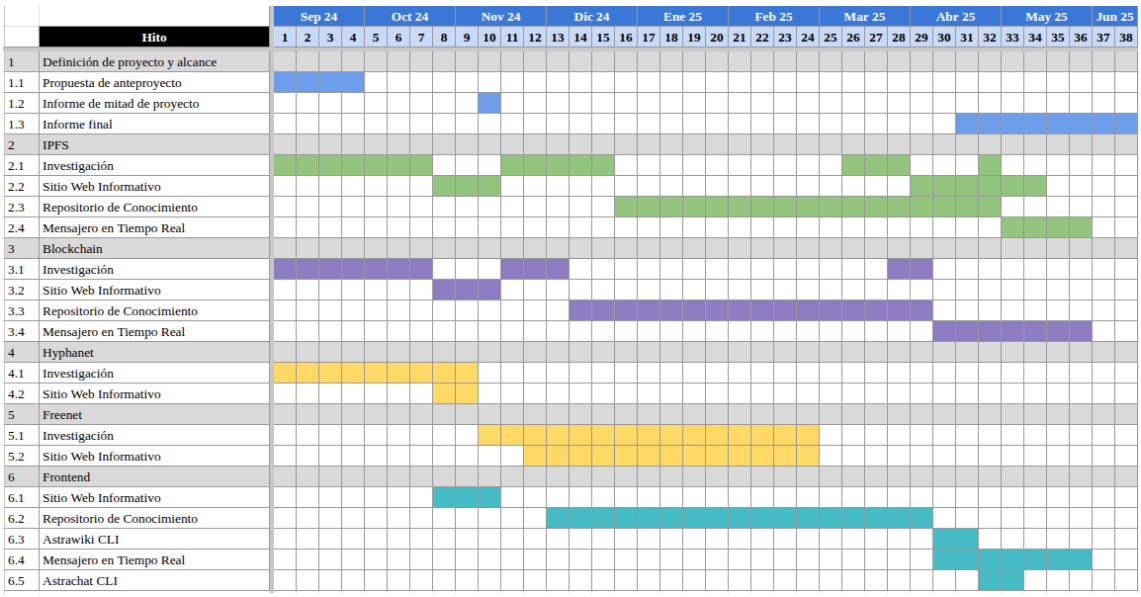

Figura 30: Cronograma real

En el cronograma real se puede ver que el desarrollo del repositorio de conocimiento en IPFS llevó más tiempo a comparación del mensajero en tiempo real. Esto fue debido a que, a medida que se fue desarrollando el repositorio de conocimiento, se fueron realizando abstracciones (como la separación en packages y AstraDB) que permitieron que el desarrollo del mensajero en tiempo real sea más sencillo y lleve menos tiempo. Algo similar sucedió para blockchain, donde notamos que los contratos para ambos casos de uso resultaron ser bastantes similares.

Además, antes del desarrollo de cada caso de uso dedicamos un tiempo a la investigación en cada ecosistema, esto incluyó: análisis de herramientas existentes, confección de *User Story Mappings* y lectura de documentación de cada ecosistema.

Por último, podemos notar el momento justo en el que dejamos de realizar el desarrollo de Hyphanet y Freenet, que fue en donde entró en juego nuestro plan de contingencia.

12. Riesgos materializados

Descripción	**Causa**	**Plan de Respuesta**	**Umbral**	**Plan de Contingencia**
Incapacidad de utilizar Hyphanet como ecosistema	Documentación desactualizada y/o API poco documentada	Tomar como referencia otras aplicaciones	Las aplicaciones de referencia no siguen un estándar	Descartar Hyphanet y reemplazar por Freenet
Incapacidad de utilizar Freenet como ecosistema	Ecosistema inestable debido al desarrollo activo	Esperar por versión estable	Versión estable para febrero de 2025	Dar de baja Freenet y agregar métricas de performance para los otros ecosistemas

Cuadro 11: Riesgos materializados

Cambio de Hyphanet a Freenet Aproximadamente un mes luego del inicio del proyecto resolvimos cambiar el tercer ecosistema elegido (Hyphanet) por su versión más moderna (Freenet). Esto fue debido a que encontramos que la documentación era escasa, los programas realizados para el ecosistema eran unos pocos y cada uno tenía una forma distinta de implementar ciertas partes. La API tampoco provee facilidades a la hora de gestionar archivos, manejo de comunicaciones, entre otras cosas que consideramos necesarias para los casos de uso.

Freenet en desarrollo Un riesgo que teníamos en cuenta eran las modificaciones que podría sufrir Freenet al estar aún en desarrollo. Esto fue de la mano con que la documentación publicada no está actualizada a la última versión.

Baja de Freenet como ecosistema Dada la promesa del equipo de Freenet de lanzar una versión estable en el corto plazo –pero que ya llevaba más de un año en ese estado– decidimos poner como límite el mes de febrero de 2025. Llegada la fecha, no hubo ningún anuncio de la versión estable (y al momento de redactar este informe tampoco lo hay) por lo que decidimos descartar el ecosistema y, en cambio, agregar métricas de performance a los otros ecosistemas.

13. Lecciones aprendidas

13.1. Tecnologías emergentes

Trabajar con tecnologías emergentes resulta un desafío al encontrarse en desarrollo constante y frecuente. Esto quiere decir que la documentación es escasa, nula o se encuentra desactualizada.

13.2. Creación de paquetes

Durante parte del desarrollo del repositorio de conocimiento, trabajamos la implementación de cada ecosistema dentro del front-end. Esta decisión nos pareció razonable luego de elegir usar un mismo front-end para ambos ecosistemas y abstraernos en una interfaz en común. Sin embargo, una vez avanzado en ambos ecosistemas, las limitaciones de esta elección se hicieron aparentes.

Por un lado, nos restringía para realizar métricas y tests manuales fácilmente, ya que cada prueba involucraba levantar un servidor local con la página y recompilar el mismo con cada cambio, además de no tener fácil acceso para pruebas automáticas.

Por otro lado, limitaba el potencial de las herramientas que estábamos desarrollando. El resultado final es mucho más versátil ya que puede acoplarse a cualquier front-end como se demostro con los front-ends alternativos, ventaja que se alinea con la filosofía de aplicaciones decentralizadas y comunitarias.

En retrospectiva desarrollar los casos de uso de forma modular desde el comienzo hubiera agilizado el proceso, e implicaría posiblemente el uso de pruebas en una etapa más temprana.

13.3. Otras lecciones

- Paralelizar las tareas por ecosistema nos ayudó a tener un referente para cada uno y no abarcar la misma información al investigar.
- Realizar reuniones semanales nos sirvió para compartir el conocimiento adquirido de los distintos ecosistemas y ponernos al tanto de en qué estaba trabajando cada uno.
- Nos abrió al mundo P2P y web descentralizada, que es una manera distinta de pensar y desarrollar aplicaciones.

14. Impactos sociales y ambientales

A continuación se realizará un análisis del posible impacto que pueda tener el uso de estas herramientas, manteniendo la distinción entre ecosistemas.

14.1. Aplicaciones en IPFS

14.1.1. Moderación y Censura

Nuestras aplicaciones desarrolladas con IPFS carecen de moderación para los artículos y chats. Esto es debido a la falta de roles, necesarios para una moderación efectiva dentro de cada comunidad. Como se vio previamente, las bases de datos que representan artículos o chats son *append-only*, y por lo tanto no hay una manera de eliminar un contenido en particular sin eliminar la base de datos.

Tampoco es sencillo eliminar la base de datos. Tanto en Astrawiki como en Astrachat, cada chat/artículo es único, y por lo tanto no se puede simplemente sobrescribir el contenido con otra base de datos. Por otro lado, la base de datos siempre estará disponible mientras otras personas la alojen.

La manera más factible de eliminar un artículo o chat por completo es a nivel de front-end, o sea, manualmente eliminando resultados de las búsquedas, y no permitiendo la carga de esos artículos o chats específicos. Esto se puede realizar siempre y cuando se tenga acceso al código fuente del front-end utilizado, y se tenga permisos para modificarlo de manera que los usuarios obtengan la nueva versión al usarlo.

Todas las ventajas y limitaciones de IPFS abarcan nuestras implementaciones. En el caso en que un individuo o entidad -como un gobierno, o un ISP- quiera censurar estos recursos, no basta con prohibir un dominio en particular, ya que IPFS es abierto y los CIDs y nombres de IPNS seguirán disponibles. Existen maneras de censurar por CID que explotan componentes de IPFS como la Distributed Hash Table, pero pueden ser mitigados. [86] Tampoco es útil inhabilitar un gateway en específico, ya que los usuarios pueden optar por usar otra gateway, o directamente acceder mediante su propio nodo de IPFS.

Por otro lado, los nodos de IPFS no son anónimos. El `PeerID` de cada nodo es permanente y se mantiene cada vez que se inicia. A la vez, se puede relacionar el `PeerID` de un nodo con una *multiaddress*, y por lo tanto, la IP pública del nodo [75]. No obstante, incluso obteniendo la IP de uno de los nodos que aloja el contenido, este suele estar disponible en múltiples nodos. Esto incluye a cualquier usuario que utilice la aplicación y gateways que la recuperen, ya que por la naturaleza de IPFS, el contenido se mantendrá en la caché del nodo por un tiempo.

La contra-cara de estas dificultades para censurar un contenido alojado en IPFS es la facilidad para confirmar que un contenido ya no está disponible. Un nodo puede saber con certeza que un contenido no está disponible simplemente intentando recuperarlo, lo cuál es difícil de lograr con servidores HTTP [25].

Además, un estudio realizado en Septiembre de 2023 [8] afirma que gran parte de los nodos presentes en la red de IPFS provienen de servicios de cloud, los cuáles son propietarios y fáciles de manipular para censurar o restringir el acceso a un contenido en específico. Entre otras estadísticas, se resalta que un 79,6 % de los nodos en la DHT provienen de datacenters, y un 80 % de los nodos que están detrás de una red con NAT, obtienen contenido mediante relays alojados en servicios de cloud.

Los servicios desarrollados con IPFS como infraestructura son muy resilientes a posibles ataques. A su vez, la dificultad para eliminar contenido no deseado presenta un posible problema, ya que se puede utilizar para alojar contenido malicioso, ilegal, o simplemente no acorde a una comunidad que aloje una wiki o chat.

Acceso a información en regiones con censura o poca conexión Un beneficio claro del uso de IPFS es su tolerancia a la partición de la red [25]. Debido a que el contenido puede ser replicado en múltiples nodos, no se puede completamente limitar el acceso y publicación de contenido. Por otro lado, una vez que un nodo en una red local o regional obtiene el contenido, este estará disponible en esa red mientras el nodo lo ofrezca, lo cuál es útil para lugares en donde la conexión a Internet es limitada, ya que para obtener dicho contenido no se requiere recuperar archivos fuera de la red local.

14.1.2. Impacto ambiental

No hay análisis realizados acerca de la utilización de energía por parte de la red de IPFS en comparación con soluciones centralizadas. Sin embargo, hay varios factores que diferencian el uso de energía de una solución centralizada con una realizada con IPFS como infraestructura.

Recuperación de contenido En un escenario óptimo, el contenido requerido está disponible en un nodo cercano. Por ejemplo, si otro usuario en un área especifico accede a un contenido, el resto de los nodos en ese área podrán recuperar el contenido desde ese nodo. En cambio, un contenido alojado en un servidor HTTP siempre requiere acceso al servidor, el cuál puede estar a mayor distancia. Esto implica un uso mayor de ISPs regionales y de *tier 1*, y en general requiere infraestructura de red de mayor capacidad.

Eficiencia Dado que parte de los nodos proveedores de contenido están ubicados en la *periferia* de la red (dispositivos móviles, computadoras de hogar, etc.), los métodos de lectura y escritura de contenido no es óptimo. Los nodos pueden estar utilizando discos duros en vez de unidades de estado sólido, o en general componentes ineficientes comparados con un servidor dedicado a proveer contenido HTTP. Esto puede resultar en un mayor uso de energía.

Escalabilidad Si bien los servicios de cloud proveen escalabilidad para variar la cantidad de máquinas dedicadas a proveer un contenido especifico, la naturaleza de IPFS hace que esta red sea incluso más optimizada. Cuando muchos nodos requieren de un contenido, se replica por la red fácilmente y sin cargas altas para ningún nodo en particular.

Realizar un análisis detallado del impacto ambiental de IPFS es uno de los trabajos futuros propuestos.

14.2. Aplicaciones en Ethereum

Se sabe que blockchain abrió un mundo de posibilidades en cuanto a la privacidad y el impacto ambiental. A continuación desarrollaremos el impacto de las aplicaciones desarrolladas en esta tecnología.

14.2.1. Moderación y Censura

Al igual que como sucede con IPFS, en Ethereum no es posible borrar un *smart contract* de la red. Esto significa que una vez creado un artículo o un chat el mismo no puede ser borrado, haciéndolo resistente a la censura.

En cuanto a la moderación de contenido por parte de gobiernos o ISP, no es posible dado que es una red distribuida y, mientras exista un nodo que tenga la información de la red, la misma va a estar disponible para todos los usuarios.

Si se trata de moderación de contenido por parte de los propios usuarios no se puede garantizar que no surjan modificaciones en los artículos o chats. Cualquier persona con la *address* del contrato y cumpliendo con la firma de los métodos del mismo puede realizar modificaciones.

14.2.2. Impacto ambiental

Desde su adopción, blockchain en general ha sido criticado por el enorme impacto ambiental y la huella de carbono al utilizar casi tanta energía eléctrica como un país entero [27]. Esto, con los años ha ido cambiando y, al día de hoy, existen blockchains *eco-friendly*[92]. En particular, la red de Ethereum pasó de utilizar *proof-of-work* a *proof-of-stake* lo cual hizo que el consumo eléctrico de los nodos disminuyera en hasta un 99 %[37]. Sin embargo, hay dudas sobre la forma de medir este impacto y compararlo con otras blockchain como Bitcoin. Además, la cantidad de nodos que se sumaron a *proof-of-stake* luego de la migración se duplicó.[68]

15. Trabajos futuros

En base al análisis realizado, se detectaron diversas lineas de trabajo futuras de las cuales se puede continuar el desarrollo.

15.1. Mejoras a los casos de uso

A continuación se proponen mejoras para los casos de uso implementados, los cuales, por temas de tiempo, no se llegaron a implementar.

15.1.1. Astrawiki-eth

Teniendo en cuenta que es el caso de uso donde mayor volumen de datos se transfieren, por lo tanto mayor costo de gas, una mejora involucra la disminución de este costo. Se podría disminuir haciendo uso de los clones de *smart contract*s según lo propuesto en el ERC-1167[72]. En resumen, lo que se propone es que haya un contrato como "plantilla" al cual se lo copia cada vez que, por ejemplo, se crea un artículo nuevo.

15.1.2. Astrachat-eth

Una de las principales mejoras que se pueden implementar sobre este caso de uso es respecto a la experiencia de uso para frontends en navegadores web. El hecho que para cada mensaje enviado sea necesario confirmar una transacción por medio de la wallet hace que la aplicación pierda el factor de *real-time*. Una posible solución a este problema es que se pueda crear un balance en un fondo común (con dinero de los usuarios) de forma que cada vez que se envíe un mensaje se tome de este fondo para pagar la transacción. Esta mejora haría más fácil la integración con otros frontends que no corren en navegadores web como *Astrachat-cli* aunque también queda investigar cómo conectar una wallet a los mismos.

Por otro lado, en base al costo por transacción (visto en la sección 10.2.4) se podría utilizar alguna red cuyos precios de gas por transacción sean menores. Una de estas redes –basada en Ethereum– es la *Ronin Network*[80][81] que a principio de este año (2025) abrió su red para que cualquier usuario pueda implementar aplicaciones en la misma.

15.2. Mejora para clusters colaborativos

Como se mencionó previamente en el análisis de IPFS, los clusters colaborativos son una alternativa para lograr la persistencia y disponibilidad de archivos, a través del "pinneo"de nodos colaborativos. La cual va muy de la mano con la filosofía de las aplicaciones comunitarias, ya que potencialmente la propia gente que utiliza la aplicación sea la que la mantenga colaborativamente. El problema es que actualmente casi no se utiliza esta alternativa, lo cual lleva a que por ejemplo, no haya una manera fácil de utilizarla, sin embargo esto tiene sus razones.

IPFS Cluster Para entender mejor por qué no se utiliza mucho esta forma de "pineoz cuales son sus limitaciones, primero hay que entender que es lo que se utiliza. IPFS cuenta con una aplicación distribuida llamada IPFS Cluster, esta funciona a la par de los nodos de IPFS y permite mantener un conjunto global de "pines", alocando inteligentemente los items a sus nodos.

Gracias a esta aplicación es posible la creación de clusters colaborativos. Los clusters colaborativos permiten que *peers* individuales, que no son de confianza, se unan a un clúster IPFS como "seguidores"(sin permiso para modificar o editar el conjunto de pines).

Para crear este cluster es necesario primero contar con algunos nodos confiables, aquellos que van a tener permitido modificar qué archivos son los que pertenecen al conjunto de "pines". Y es esta nuestra primera limitación, ya que no se puede permitir que cualquier nodo pueda modificar este conjunto, entonces es necesario contar con nodos propios que tengan ese rol.

Otra cosa muy importante que se puede configurar sobre el cluster es el *replication factor*, este ajuste permite indicar a cuantos nodos se va a replicar cada archivo. Esto es muy importante porque permite que no sea obligatorio replicar la totalidad de los archivos en todos los nodos.

Sin embargo existe una limitación muy importante, IPFS Cluster no verifica si los seguidores realmente están "pineando.º no el contenido, ni cuánto espacio tienen ni otras métricas. Confía en todo lo que dicen. Eso significa que un seguidor malintencionado siempre puede fingir que tiene suficiente espacio, que se le asignen pines o fragmentos, pero al final no "pinea"nada. Un grupo formado por voluntarios aleatorios y que no son de confianza no puede usar, por ejemplo, factor de replicación de tres, porque esos 3 miembros podrían simplemente estar fingiendo "pinearçosas.

Es por eso que el escenario que mejor funciona es dejando que todos "pineen"todo y esperar que algunos de los *peers* no sean maliciosos.

Justamente de esa manera es como funcionan los clusters públicos actualmente, visibles en la página de IPFS[50], donde hay algunos que piden gigabytes, y hasta algunos terabytes de información, para poder colaborar. Esto es algo totalmente inviable para que la gente pueda colaborar hasta un cierto almacenamiento, limitando mucho la posibilidad de ditribución de colaboradores.

Deployado, seguimiento y descubrimiento Habiendo hablado ya de las limitaciones más técnicas que se prensentan al usar cluster colaborativos, hay otro tipo de limitación que es el la falta de facilidad a lo hora de deployar, seguir y descubrir aplicaciones colaborativas que usen IPFS cluster.

En primer lugar hablemos del deployado, actualmente si se quiere deployar la aplicación en un cluster colaborativo la única forma que existe es hacerlo de forma manual y crear un servicio que se ocupe de actualizar el conjunto de "pinesçuando sea necesario actualizarse. Si bien es factible, no es algo cómodo para todos y limita a decidirse ir por este camino. Lo ideal sería, que exista un servicio, parecido al ofrecido en los servicios de *pinning*, como es el caso de Fleek[30], donde al actualizar el repositorio ya se deployen los cambios. Facilitando asi su deployado en el cluster colaborativo. Se podría pensar como un servicio que corra sobre los nodos *trusted* donde, cuando un nodo lider detecte que se hizo un cambio, se "buildee"la página web por ejemplo, y se despliegue en el cluster comunitario.

Parecido a lo anterior, alguien que quiera colaborar necesita ejecutar comandos manualmente que para mucha gente no sabe lo que está haciendo, lo cual también termina desincentivando y generando que menos gente decida por colaborar.

Por último no existe una manera fácil de presentar y descubrir nuevos o existentes proyectos comunitarios que la gente pueda optar por colaborar. Actualmente el único lugar donde prensentan proyectos públicos para colaborar es la página de IPFS[50], que ademas se ser bastante simple, depende de ellos si agregan un nuevo proyecto para presentar en esa página. Donde lo ideal sería que puedan publicarse nuevos proyectos junto a una descripción y que la gente pueda conocerlos y optar por colaborar de una manera mucho más sencilla.

Conclusión Con esto concluimos que se presenta la oportunidad para una linea de trabajo futura focalizada en la mejora para clusters colaborativos.

En primer lugar una mejora técnica donde se mofique o se contruya por encima de IPFS cluster, que permita principalmente que en un clúster comunitario se asegure el "pineo"de información sin necesidad de tener que obligar a todos los colaboradores a "pinear"la totalidad de los archivos.

En segundo lugar, la creación de una plataforma o aplicación, que permita la facilitación del deployado, seguimiento y descubrimiento de aplicaciones colaborativas.

15.3. Análisis del consumo de energía en la red de IPFS

Como se mencionó anteriormente, no hay análisis detallados del uso de energía en la red de IPFS. Un posible trabajo involucra calcular una huella de carbono aproximada, el consumo promedio de energía, entre otras métricas. De ello puede desprenderse un análisis comparativo entre diferentes ecosistemas como Blockchain, y cloud hosting tradicional.

15.4. Blockchain para aplicaciones comunitarias

En la búsqueda de una blockchain comunitaria nos encontramos con un servicio llamado *Filecoin*[59] el cual provee de incentivos monetarios para aquellos que cedan espacio en sus discos a guardar archivos de otras personas. Es abierto para que cualquier persona pueda unirse a la red peer-to-peer que proveen. Funciona por arriba de IPFS, utiliza tecnologías de Blockchain para los incentivos y dar garantía que los datos están realmente guardados en los nodos. Nuestra propuesta

de trabajo futuro es que se cree una tecnología basada en Blockchain que no necesite del incentivo monetario para funcionar y que además permita la ejecución de código.

15.5. Análisis de Freenet como ecosistema

Si en un futuro Freenet finalmente sale en una versión estable, creemos que sería un buen ecosistema al cual realizar un análisis como el de este trabajo y hasta más en profundidad con otro tipo de aplicaciones distribuidas y descentralizadas.

16. Conclusiones

16.1. Conclusión del análisis

16.2. Conclusión general

17. Referencias

[1] *ACME*. (s.f.). https://letsencrypt.org/how-it-works/
[2] *Anatomy of an IPNS name*. (s.f.). https://docs.ipfs.tech/concepts/ipns/#anatomy-of-an-ipns-name
[3] *Astrachat ETH*. (s.f.). https://github.com/bitxenia/astrachat-eth
[4] *Astrachat IPFS*. (s.f.). https://github.com/bitxenia/astrachat
[5] *Astrawiki ETH*. (s.f.). https://github.com/bitxenia/astrawiki-eth
[6] *Astrawiki IPFS*. (s.f.). https://github.com/bitxenia/astrawiki
[7] *AutoTLS*. (s.f.). https://blog.libp2p.io/autotls/
[8] Balduf, L., Korczyński, M., Ascigil, O., Keizer, N. V., Pavlou, G., Scheuermann, B., & Król, M. (2023). The Cloud Strikes Back: Investigating the Decentralization of IPFS. *Proceedings of the 2023 ACM on Internet Measurement Conference*, 391-405. https://doi.org/10.1145/3618257.3624797
[9] Batt, S. (2021). Your Files for Keeps Forever with IPFS. *Opera Blog*. https://blogs.opera.com/tips-and-tricks/2021/02/opera-crypto-files-for-keeps-ipfs-unstoppable-domains/
[10] Benet, J. (2014). IPFS - Content Addressed, Versioned, P2P File System. *IPFS Project site*. https://github.com/ipfs/papers/raw/master/ipfs-cap2pfs/ipfs-p2p-file-system.pdf
[11] Bondy, B. (2024). IPFS Support in Brave. *Brave Blog*. https://brave.com/blog/ipfs-support/
[12] Brnakova, J. (s.f.). *Game decommissioning: When beloved games shut down*. https://www.revolgy.com/insights/blog/game-decommissioning-when-beloved-games-get-shut-down-and-online-worlds-disappear
[13] *Censorship of Wikipedia*. (s.f.). https://en.wikipedia.org/wiki/Censorship_of_Wikipedia
[14] CLARKE, I. (1999). 'Freenet : A distributed anonymous information storage and retrieval system,' white paper of The free Network Project. *http://freenetproject.org/index.php?page=index*. https://cir.nii.ac.jp/crid/1572824499747330944
[15] *Collaborative Clusters*. (s.f.). https://ipfscluster.io/documentation/collaborative/
[16] *Collaborative clusters setup*. (s.f.). https://ipfscluster.io/documentation/collaborative/setup/
[17] *Configuration Reference*. (s.f.). https://ipfscluster.io/documentation/reference/configuration
[18] *Content Addressing*. (s.f.). https://docs.ipfs.tech/concepts/glossary/#content-addressing
[19] *Daemon (computing)*. (s.f.). https://en.wikipedia.org/wiki/Daemon_(computing)
[20] *Data Differencing*. (s.f.). https://en.wikipedia.org/wiki/Data_differencing
[21] *Differences between js-libp2p in Node.js and browser*. (s.f.). https://github.com/libp2p/docs/blob/master/content/guides/getting-started/webrtc.md#differences-between-js-libp2p-in-nodejs-and-browser
[22] *Distributed Hash Tables (DHTs)*. (s.f.). https://docs.ipfs.tech/concepts/dht

[23] *Distributed wikipedia mirror.* (s.f.). https://github.com/ipfs/distributed-wikipedia-mirror/tree/cbab5fc09c622249d5433abddb0a5976e2a51875?tab=readme-ov-file#goal-2-fully-read-write-wikipedia-on-ipfs
[24] *DNSLink.* (s.f.). https://docs.ipfs.tech/concepts/dnslink/#dnslink
[25] Doan, T. V., Psaras, Y., Ott, J., & Bajpai, V. (2022). Towards Decentralised Cloud Storage with IPFS: Opportunities, Challenges, and Future Directions. https://arxiv.org/abs/2202.06315
[26] *ENS.* (s.f.). https://ens.domains/
[27] *Ethereum blockchain produced equivalent of Honduras' annual emissions before upgrade - study.* (s.f.). https://www.reuters.com/sustainability/ethereum-blockchain-produced-equivalent-honduras-annual-emissions-before-upgrade-2023-12-01/
[28] *Fees.* (s.f.). https://support.ens.domains/en/articles/7900605-fees
[29] *Filebase.* (s.f.). https://filebase.com/
[30] *Fleek.* (s.f.). https://fleek.xyz/docs/platform/hosting/
[31] *Freenet.* (s.f.). https://freenet.org
[32] *Gateway Proxy.* (2025). https://github.com/ethersphere/gateway-proxy
[33] *Hardhat.* (s.f.). https://hardhat.org/
[34] *Helia: IPFS node implementation in Javascript.* (s.f.). https://github.com/ipfs/helia
[35] *Hole Punching.* (s.f.). https://docs.libp2p.io/concepts/nat/hole-punching/
[36] *Host a single-page website with IPFS Desktop.* (s.f.). https://docs.ipfs.tech/how-to/websites-on-ipfs/single-page-website/#set-up-a-domain
[37] *How Green Is Ethereum?* (s.f.). https://www.investopedia.com/how-green-is-ethereum-2-0-6666266
[38] *How to facilitate sharding on collaborative clusters?* (s.f.). https://discuss.ipfs.tech/t/how-to-facilitate-sharding-on-collaborative-clusters/18052
[39] *HTTP API for IPFS Cluster.* (s.f.). https://ipfscluster.io/documentation/reference/api/
[40] *HTTP API for Kubo.* (s.f.). https://docs.ipfs.tech/reference/kubo/rpc/
[41] Hurtado, J. S. (s.f.). *Qué es Blockchain y cómo funciona la tecnología Blockchain.* Consultado el 25 de septiembre de 2024, desde https://www.iebschool.com/blog/blockchain-cadena-bloques-revoluciona-sector-financiero-finanzas
[42] *Hyphanet.* (s.f.). https://www.hyphanet.org/index.html
[43] *Hyphanet Plugin - Freemail.* (s.f.). https://github.com/hyphanet/plugin-Freemail
[44] *Hyphanet Plugin - Freetalk.* (s.f.). https://github.com/hyphanet/plugin-Freetalk
[45] *Hyphanet Plugin - jSite.* (s.f.). https://github.com/hyphanet/wiki/wiki/jSite
[46] *Hyphanet Plugin - Sone.* (s.f.). https://github.com/Bombe/Sone
[47] *Hyphanet Plugin - Web of Trust.* (s.f.). https://github.com/hyphanet/plugin-WebOfTrust
[48] *Ink.* (s.f.). https://github.com/vadimdemedes/ink
[49] *InterPlanetary Name System.* (s.f.). https://docs.ipfs.tech/concepts/ipns
[50] *IPFS Cluster - Collaborative Clusters.* (s.f.). https://collab.ipfscluster.io/#list-of-clusters
[51] *IPFS Content Identifiers.* (s.f.). https://docs.ipfs.tech/concepts/content-addressing/#what-is-a-cid
[52] *IPFS Garbage Collector.* (s.f.). https://docs.ipfs.tech/concepts/persistence/#garbage-collection
[53] *IPFS Gateway.* (s.f.). https://docs.ipfs.tech/concepts/ipfs-gateway/
[54] *IPFS Pinning services.* (s.f.). https://docs.ipfs.tech/concepts/persistence/#pinning-services
[55] *ipfs-cluster-follow.* (s.f.). https://ipfscluster.io/documentation/reference/follow/
[56] *IPNS Record and Protocol.* (s.f.). https://specs.ipfs.tech/ipns/ipns-record/
[57] Janardhan, S. (s.f.). *More details about the October 4 outage.* https://engineering.fb.com/2021/10/05/networking-traffic/outage-details/
[58] *Kubo: IPFS node implementation in Go.* (s.f.). https://docs.ipfs.tech/install/command-line/
[59] Labs, P. (2017, 19 de julio). *Filecoin: A Decentralized Storage Network.* https://www.filecoin.io/filecoin.pdf
[60] *libp2p.* (s.f.). https://libp2p.io/
[61] *Limo.* (s.f.). https://eth.limo/
[62] *Markdown.* (s.f.). https://en.wikipedia.org/wiki/Markdown
[63] *Metamask.* (s.f.). https://metamask.io/

[64] Mittal, N. (2007). *Timestamping Messages and Events in a Distributed System using Synchronous Communication* [Tesis doctoral, University of Texas]. https://users.ece.utexas.edu/~garg/dist/dc07-timestamp.pdf
[65] *Multiaddr*. (s.f.). https://multiformats.io/multiaddr/
[66] *Node.js*. (s.f.). https://nodejs.org/es
[67] *An O(ND) Difference Algorithm and Its Variations*. (s.f.). https://neil.fraser.name/writing/diff/myers.pdf
[68] *One Year After The Merge: Sustainability Of Ethereum's Proof-Of-Stake Is Uncertain*. (s.f.). https://www.forbes.com/sites/digital-assets/2023/10/11/one-year-after-the-merge-sustainability-of-ethereums-proof-of-stake-is-uncertain/
[69] *OrbitDB*. (s.f.). https://orbitdb.org
[70] *The Origins of Swarm*. (2015). https://blog.ethswarm.org/hive/2024/the-origins-of-swarm
[71] Petar Maymounkov, D. M. (2023). Kadmelia: A Peer-to-peer Information System Based on the XOR Metric. *Stanford Secure Computer Systems Group*. https://www.scs.stanford.edu/~dm/home/papers/kpos.pdf
[72] *Peter Murray (@yarrumretep), Nate Welch (@flygoing), Joe Messerman (@JAMesserman), "ERC-1167: Minimal Proxy Contract," Ethereum Improvement Proposals, no. 1167, June 2018. [Online serial]. Available: https://eips.ethereum.org/EIPS/eip-1167*. (s.f.). https://eips.ethereum.org/EIPS/eip-1167
[73] *Pinata*. (s.f.). https://pinata.cloud/ipfs
[74] *Pinning*. (s.f.). https://docs.ipfs.tech/concepts/glossary/#pinning
[75] *Privacy and Encryption*. (s.f.). https://docs.ipfs.tech/concepts/privacy-and-encryption
[76] *Public Gateway Checker*. (s.f.). https://ipfs.github.io/public-gateway-checker/
[77] *React*. (s.f.). https://react.dev/
[78] *The React Framework for the Web*. (s.f.). https://nextjs.org/
[79] *Repositorio del nodo confiable*. (s.f.). https://github.com/bitxenia/astrawiki-web-trusted-peer
[80] *Ronin Network*. (s.f.). https://roninchain.com/
[81] *Ronin Network Whitepaper*. (s.f.). https://docs.roninchain.com/basics/white-paper
[82] *rubix-documents*. (s.f.). https://github.com/rubixvi/rubix-documents
[83] *Server Components*. (s.f.). https://react.dev/reference/rsc/server-components
[84] *Small-World Routing*. (s.f.). https://en.wikipedia.org/wiki/Small-world_routing
[85] *SQLite*. (s.f.). https://www.sqlite.org/
[86] Sridhar, S., Ascigil, O., Keizer, N., Genon, F., Pierre, S., Psaras, Y., Rivière, E., & Król, M. (2023). Content censorship in the interplanetary file system. *arXiv preprint arXiv:2307.12212*.
[87] *Swarm CLI*. (2025). https://github.com/ethersphere/swarm-cli
[88] Team, S. (2021). Swarm; Storage and communication infrastructure for a self-sovereign digital society. *URL https://www.ethswarm.org/swarm-whitepaper.pdf*.
[89] *A Universally Unique IDentifier (UUID) URN Namespace*. (s.f.). https://www.rfc-editor.org/rfc/rfc4122
[90] *WebRTC*. (s.f.). https://docs.libp2p.io/guides/getting-started/webrtc/
[91] *The WebSocket Protocol*. (s.f.). https://www.rfc-editor.org/rfc/rfc6455
[92] *What is a Green Blockchain? Eco-Friendly Tech*. (s.f.). https://www.casper.network/get-started/a-brief-guide-to-green-blockchain-technology
[93] *Wikipedia Statistics: Edits*. (s.f.). https://en.wikipedia.org/wiki/Wikipedia:Statistics#Edits
[94] Wood, G., et al. (2014). Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, *151*(2014), 1-32.

18. Anexos

18.1. Arquitectura del repositorio de conocimiento

La arquitectura del repositorio de conocimiento fue diseñada de manera independiente del ecosistema utilizado. Se tomó esta decisión para proveer una interfaz común, independiente de su implementación en IPFS y Ethereum. De esta manera, una aplicación puede trabajar sobre una

API común e intercambiar implementaciones fácilmente. Esto es útil para la implementación del front-end del repositorio, como se verá más adelante.

En sí, la arquitectura consiste de una o varias *wikis*, dependiendo de la implementación. Cada wiki se identifica con un nombre único que representa un grupo de artículos en conjunto. Cada wiki se representa con una lista de artículos, los cuales son identificados por sus nombres —también únicos.

Un artículo se compone de un nombre único y no vacío y su contenido. Si bien nuestra implementación utiliza *markdown* [62] como formato del contenido, el tipo de contenido es arbitrario e indiferente para la wiki, cualquier tipo de archivo es admisible. Únicamente será tenido en cuenta en el momento en el que la aplicación que interactúe con el contenido lo interprete. En nuestro caso, markdown representa una manera sencilla de enriquecer un texto, lo cual lo hace apropiado para un repositorio de conocimiento comunitario.

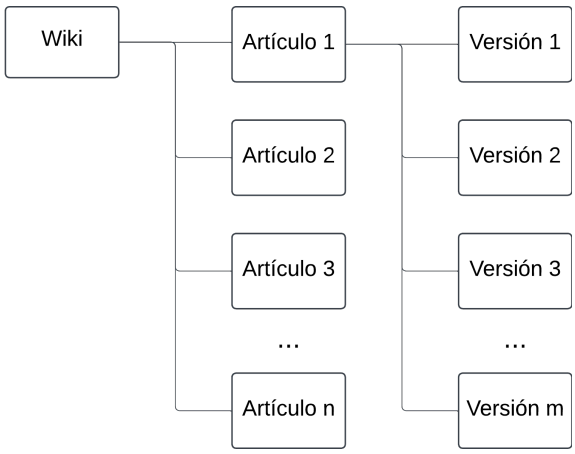

Figura 31: Arquitectura general del repositorio de conocimiento.

18.1.1. Representación de un artículo

Internamente, los artículos pueden editarse en el tiempo, y un usuario debe obtener siempre la última versión. Sin embargo, se debe poder ver versiones anteriores si así lo desea el usuario, por lo que guardar la última versión únicamente no es válido. Es por esto que se decidió representar a un artículo como un nombre, y un conjunto de versiones. Cada versión representa una modificación hecha al artículo. Un artículo nuevo tendrá una única versión, y al hacerle una edición tendrá dos.

Esta iteración de la arquitectura permite a los usuarios ver el historial de versiones, y acceder a cualquiera de ellas. Pero guardar todo un artículo por cada edición puede ser costoso en concepto de almacenamiento. Rara vez un artículo es modificado en su totalidad [93]. En cambio, las ediciones que puede sufrir un artículo se deben en su mayoría a la inclusión de contenido nuevo, o la modificación de una sección en particular. Guardar el contenido completo en casos donde los cambios son mínimos resulta ineficiente, y es particularmente negativo en aplicaciones peer-to-peer, donde el contenido se almacena en el lado del usuario.

Generación de patches Como solución para minimizar la información redundante en cada versión, se decidió usar el tipo de algoritmo de *data differencing* [20], que consiste en almacenar únicamente los cambios hechos en una versión con respecto a su antecesora, de forma similar a *Git*. Esta lista de cambios forma un *patch*.

Un patch se obtiene al aplicar un algoritmo que reciba el contenido anterior, y el contenido nuevo, y devuelva un patch que pueda ser utilizado posteriormente para reconstruir el contenido. Si bien hay varios algoritmos que logran este objetivo, se decidió utilizar el algoritmo de diff de Myers [67], ya que ofrece un punto medio entre compresión, y cómputo necesario para compilar el

texto. Este algoritmo es el que utiliza la biblioteca diff-match-patch, creada por Google, que ofrece una interfaz sencilla para lograr generar patches y luego compilar texto en base a ellos.

Calculando los patches de la nueva versión con respecto a la anterior, reducimos el espacio necesario para almacenar un artículo con todas sus versiones. Hay casos en los que un patch puede tener un tamaño mayor al de simplemente almacenar una copia del nuevo contenido. Sin embargo, estos casos no son frecuentes en usos comunes y decidimos despreciarlos, aunque puede ser una posible mejora para optimizar más aún el espacio utilizado.

Por otra parte, ya que cada versión no tiene toda la información, se necesitan todas las versiones anteriores para compilar el contenido resultante. Esto puede ser una desventaja en archivos con muchas ediciones, ya que requieren de mayor cómputo para compilar, específicamente `O(n)`, donde `n` es la cantidad de versiones que tiene el artículo. Para mejorar esto, una posible mejora consiste en fijar un número máximo de patches consecutivos que, al superarse, indica que la siguiente versión debe contener el contenido completo. De esta manera se acota la cantidad de iteraciones que puede tomar la compilación de un artículo.

La nueva iteración de la arquitectura apunta a tener una lista de versiones, que a su vez contengan el patch con las diferencias con la versión anterior. Pero como se verá en la siguiente sección, esta solución no es lo suficientemente resiliente.

18.1.2. Concurrencia

En las redes peer-to-peer es común que la replicación de contenido lleve un tiempo no despreciable. Esto puede generar problemas, ya que dos nodos pueden publicar contenido que entre en conflicto, de la misma manera que dos cambios pueden generar un *merge conflict* en Git. En el mejor de los casos, ambos nodos publican contenido que no entra en conflicto entre sí, y por lo tanto la compilación se puede realizar sin problemas. En el caso contrario, puede dejar el artículo en un estado inválido, imposible de compilar y por lo tanto, inaccesible.

Yendo a nuestro caso de uso, el problema planteado previamente puede entrar en conflicto de dos maneras:

1. Al crear un articulo
2. Al editar un artículo

Por lo tanto, es necesario una solución que mitigue estos problemas.

Para el caso de la creación de dos artículos con en mismo nombre, se delega en cada ecosistema la responsabilidad de detectar estos casos y actuar en consecuencia. La implementación de cada ecosistema contiene más información para tomar una decisión optimizada al respecto, la cuál se verá en secciones posteriores. En cambio, el problema de la edición de un artículo de forma concurrente puede ser resuelto a nivel arquitectura.

18.1.3. Árbol de versiones

Hasta ahora, cada versión se agrega a una lista de versiones, y se asume que cada versión se basa en la anterior. Cuando ocurre un conflicto de concurrencia entre versiones, esto no aplica por las razones dadas previamente. Por lo tanto, es necesario que cada versión tenga un ID para que las versiones posteriores puedan mantener registro de su versión padre, es decir, la versión sobre la que se basa. Todas las versiones tienen padre, excepto la versión inicial. Cuando dos versiones se crean en base a una misma versión padre, se genera una bifurcación. Por lo tanto, las versiones pasan a formar un árbol.

En arquitecturas tradicionales, la generación de IDs suele hacerse por la misma base de datos, de manera incremental. Esto no genera problemas ya que el servidor decide en que orden tomar las versiones. En una aplicación distribuida, esto no es conveniente, ya que dos nodos pueden generar el mismo ID por la misma razón por la que pueden generar versiones conflictivas. La solución

elegida fue usar `UUIDs` [89], que se generan sin necesidad de centralizar la decisión, y tienen una probabilidad de colisión casi nula.

Una problemática que trae tener distintas ramas de versiones es que la compilación del contenido deja de ser trivial, ya que hay múltiples posibles últimas versiones, dependiendo de la rama que se elija. Para resolver esto es necesario definir una heurística que elija la rama principal del artículo de forma determinista.

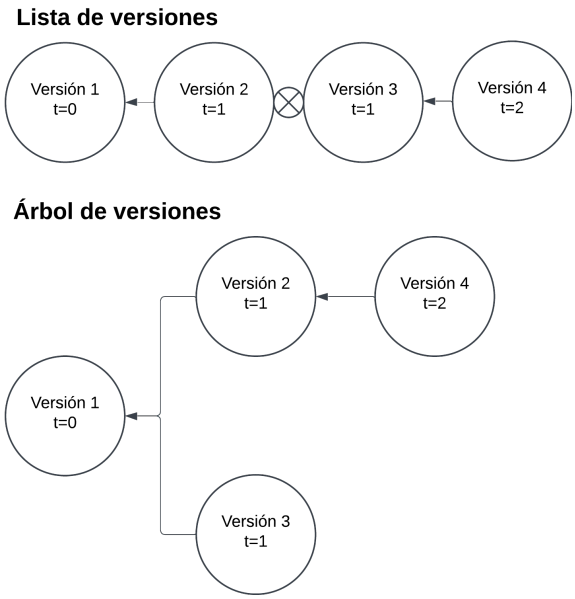

Figura 32: Ambas maneras de almacenar versiones. En este caso, `t` es un momento en el tiempo en el que todas las bases de datos están sincronizadas. En el primer caso, cuando `t=1` en ambas versiones, significa que hay un conflicto ya que ambas se basan en la misma versión padre. En el árbol, ese problema se mitiga.

Elección de rama principal Una de las decisiones tomadas respecto al árbol de versiones, es que los usuarios sólo puedan modificar la última versión "principal", es decir; un usuario no podrá editar en base a una versión arbitraria. Teniendo en cuenta esto, es fácil deducir que las ramas no principales o secundarias tendrán a lo sumo una versión en la mayoría de los casos. Esto es debido a que los demás usuarios que no hayan causado la colisión elegirán una rama principal, por lo tanto las demás ramas dejarán de recibir versiones nuevas. Elegir la rama de mayor cantidad de versiones será parte de la heurística para elegir la rama principal.

Sin embargo, esta decisión no es exhaustiva, ya que puede haber dos ramas con la misma longitud. De hecho, este caso es muy común debido a que dos usuarios que crean una nueva versión en base a la misma versión padre generan dos ramas del mismo tamaño. Para tomar una decisión que sea determinista e igual en todos los nodos, se decidió tomar la antigüedad de la última versión de cada rama como parámetro para la heurística. La rama mas antigua será considerada la rama principal. La antigüedad de una rama es decidida por una marca de tiempo o *timestamp* grabada en el mismo nodo que crea la rama, y es enviado como parte de la versión a los demás nodos. Esto trae una desventaja en cuánto a seguridad, ya que un nodo puede falsificar un timestamp y así tener prioridad siempre, pero es mejorable utilizando algoritmos para crear timestamps en un sistema distribuido [64].

```
type VersionID = string;

type Version = {
  id: VersionID;
  date: string;
  patch: Patch;
  parent: VersionID | null;
};
```

Figura 33: Propiedades de una `Version`

Una vez elegida la versión principal, se considera como rama principal a todas las versiones que entren en la cadena de padres, empezando por la versión principal hasta llegar a la versión raíz. De esta forma, sabemos que la rama principal siempre se podrá compilar, generando una red mucho más resiliente a cambios concurrentes. Por último, las versiones que no sean parte de la rama principal también podrán visualizarse. Se tomó esta decisión para permitir que los usuarios que hayan subido una versión por fuera de la rama principal puedan ver el contenido que editaron y, en todo caso, editar la nueva versión principal con el mismo.

18.1.4. Arquitectura del mensajero en tiempo real

La arquitectura para un mensajero es mucho más sencilla que la del repositorio de conocimiento. Consta de un grupo de chats, que a su vez contienen todos los mensajes. Cada mensaje contiene el texto enviado, el usuario que lo envió, un *timestamp* del momento de su envío, entre otras cosas.

Aliases En las implementaciones que se verán posteriormente, se determina que la manera de identificar a un usuario será mediante un ID similar a un hash en ambos casos. Esto es común para sistemas descentralizados, pero vuelve difícil diferenciar los usuarios en un chat. Para solucionar esto, se decidió agregar un alias que identifique al usuario, que no tenga que ser único y pueda ser cambiado. Su implementación varía dependiendo del ecosistema, pero en ambos es parte del objeto `ChatMessage` que almacena cada chat.

Respuestas Se decidió identificar cada mensaje mediante un identificador, de manera tal que se pueda indicar el "padre"de un mensaje, o sea, el mensaje al que se le está respondiendo.

```
export type ChatMessage = {
  id: string;
  parentId: string;
  sender: string;
  senderAlias: string;
  message: string;
  timestamp: number;
};
```

Figura 34: Propiedades de un `ChatMessage`